ENERGY EFFICIENT PARALLEL AND DISTRIBUTED SIMULATION

A Dissertation
Presented to
The Academic Faculty

by

Aradhya Biswas

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
August 2019

COPYRIGHT © 2019 BY ARADHYA BISWAS

ENERGY EFFICIENT PARALLEL AND DISTRIBUTED SIMULATION

Approved by:

Dr. Richard M. Fujimoto, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Margaret L. Loper
Georgia Tech Research Institute
Georgia Institute of Technology

Dr. Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard W. Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Michael P. Hunter
School of Civil and Environmental Engineering
Georgia Institute of Technology

Date Approved: 19 July 2019

To my parents, my brother and all my teachers.

ACKNOWLEDGEMENTS

This work wouldn't have been possible without the support and help of a large group of people. First and foremost, I would like to acknowledge the help, support and guidance provided by my doctoral advisor, Dr. Richard Fujimoto. I am immensely grateful for his time and patience throughout this journey. In spite of his busy schedule and responsibilities, he always made time for my questions, my research, and me. I am genuinely and exceedingly thankful for it.

During the course of this work, I had the chance to work with Dr. Michael Hunter and Dr. Rich Vuduc. I am thankful for their knowledgeable advice and support all the way since my initial days as a graduate student. I am also thankful for their time to serve on all my doctoral committees. I would also like to thank Dr. Ümit Çatalyürek and Dr. Margaret Loper for their time to serve on my thesis committee and their constructive comments and observations.

Next, I would like to thank my fellow graduate students for making this a pleasant and enjoyable journey. I am especially thankful to the cohort of Fall 2015 CSE PhD students, who embarked on this journey with me, and my collaborators, Sabra Neal, Philip Pecher and Mark Jackson for their honest feedbacks and our long-winded discussions. Of the numerous campus organizations that further made this process supercalifragilisticexpialidocious, I would like to specifically thank GT cycling team for the weekend escapes.

This would be incomplete without a mention of the awesome staff of the School of Computational Science and Engineering. I would like to heartily thank them for their warm and welcoming disposition. They often went out of their ways to help me out whenever I needed it. It is impossible to name all of them here, but I would like to acknowledge the help and support I received from Arlene Washington-Capers and Holly Rush, whom I have known since my undergraduate internship days at the school.

Before moving on, I would like to thank and acknowledge all my teachers. Ones who taught me my first letters all the way to the ones who taught me the concepts of parallel and distributed computing.

Finally, I would like to acknowledge my family for being there for me at every step of the way. Significantly, my grandmothers for setting up the stage for my success and my cousin, Sovan Biswas, for being a constant source of inspiration and for leading the way. Though this work is dedicated to them, I would like to specifically acknowledge my parents, Ashis and Rina Biswas, and my brother and partner in crime, Aditya Biswas, for their unconditional love, support and encouragement. I owe my deepest and most sincere gratitude to them, as it is through their guidance, patience, sacrifices, hardships and trust that I could be at this stage today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

LP	Logical Process
IoT	Internet of Things
DDDAS	Dynamic Data Driven Application Systems
G-RTI	Green Runtime Infrastructure
PADS	Parallel and Distributed Simulation
CMB	Chandy-Misra-Bryant
YAWNS	Yet Another Windowing Network Simulator
FIFO	First-In-First-Out
HLA	High Level Architecture
GPU	Graphical Processing Unit
HPC	High Performance Computing
P2P	Peer-to-Peer
VANET	Vehicular Ad-Hoc Network

SUMMARY

As computing becomes more integrated into our daily lives, new challenges emerge as computations interact with our surroundings in new and unprecedented ways. One of these challenges is the power and energy consumed by computations and communications. In large cloud-based computing systems, power consumption is a major concern because it forms the largest portion of the operating costs of data centers. In mobile systems energy consumption directly impacts battery life. This work focuses on understanding and minimizing power and energy consumption in parallel and distributed execution of discrete event simulations, an area not extensively studied in the past.

This body of work begins by describing a series of empirical experiments to assess the energy and power consumption of distributed simulations. A unique aspect of parallel and distributed simulations is the need for a synchronization algorithm to ensure the concurrent execution yields the same results as a sequential execution. This study characterizes the energy consumption of widely used synchronization algorithms and demonstrates that synchronization can incur a significant power overhead.

We then propose a model and methodology to create energy profiles to empirically determine the energy consumed by different aspects of a distributed simulation system. Creating energy profiles is not straightforward because high precision, low overhead energy measurement mechanisms may not be available, and it is not straightforward to determine the amount of energy consumed by different concurrently operating hardware components. The methodology is exercised by measuring the energy consumed by different aspects of the distributed simulation system. Techniques to

optimize costs associated with distributed execution of simulations are evaluated and demonstrated to yield significant benefits.

Noting that synchronization can be a significant component of energy consumption in distributed simulations, a theoretical concept termed *zero-energy synchronization* is proposed as a goal to achieve the minimum amount of energy required to execute a given distributed simulation program. An algorithm based on an oracle is proposed as a means to determine this minimum amount of energy required for any distributed simulation code, and a practical implementation developed.

A synchronization algorithm termed Low Energy YAWNS (LEY) is proposed to reduce the energy consumption of distributed simulations. LEY represents the first attempt to design a synchronization algorithm for energy efficiency. It is shown that LEY, in principle, can achieve zero-energy synchronization for a large class of distributed simulation applications.

To analytically study energy consumption of synchronization algorithms a model is proposed. This model is used to analyze and compare energy consumption of simulations synchronized with various conservative synchronization algorithms. Analytical and empirical measurements of an implementation of LEY demonstrate that it can yield energy consumption much less than a comparable synchronization algorithm that does not consider minimizing energy consumption as a goal, and consumes energy approaching that obtained using an oracle.

Another avenue for improving the energy and power efficiency of a simulation is to use specialized hardware (or a group of them) such as Graphical Processing Units

(GPU) or custom Field Programmable Gate Arrays (FPGA). These computing elements, although limited in their ability when compared to a traditional CPU, are more energy efficient per unit computation. We propose and empirically analyze a recurrence relation for simulation of G/G/1 queueing networks. The proposed recurrence, in addition to MIMD and SISD (i.e., CPU) based general purpose distributed computing environments, can also be efficiently implemented on SIMD systems (such as GPUs). The simplicity of the recurrence allows it to be directly implementable using primitives provided by all major parallel processing libraries. This not only eases the development process but also allows use of library primitives, which are generally heavily optimized and scalable. In addition, it also opens avenues for adapting the simulation for custom hardware as they become available.

To further improve the performance and energy efficiency of the proposed recurrences and any series of parallel prefix scans with the distributive operator property, composition of parallel prefix scans is proposed. The composition of parallel prefix scan is described and its application to queuing network simulation is presented. The performance improvement achieved by composition of parallel prefix scans in the proposed recurrence for queueing network simulation is demonstrated using a 2D toroidal queueing network simulation.

Finally, the Green Runtime Infrastructure (G-RTI) middleware is described to provide a platform for creating energy-efficient distributed simulations for dynamic data driven applications systems (DDDAS) on heterogeneous platforms. G-RTI supports simulation, emulation and deployment of DDDAS with a wide variety of heterogeneous and energy-constrained components, including as-thin-as-required clients.

1 INTRODUCTION

Applications of parallel and distributed computing range from purely academic explorations to commercial applications. Faster execution and decentralization are two goals that have historically driven its use in a variety of applications.

An important application of parallel and distributed computing is simulation. Simulations are computationally expensive and form the bulk of the computing done by some of the largest super computing systems. Consequently, simulations have historically been directly or indirectly a major candidate for efforts to achieve faster execution times. In addition to the direct implication of faster turn-around times, faster execution can also enhance the reliability of results produced by the simulations by allowing for more repetitions and/or higher fidelity models.

Since the breakdown of Dennard scaling parallel and distributed computing have taken on added importance in efforts to sustain exponential increases in compute capability. This change in the computing industry was marked at the Intel Developers Forum in 2004 by the then-president of Intel Paul Otellini's remark: "We are dedicating all of our future product development to multicore designs ... This is a sea change in computing". The recent ending of Moore's law (Hennessy and Patterson 2018) has further pushed the field forward.

A concurrent rise in networking technologies has made computing in a distributed array of systems more amenable to a new spectrum of applications and use cases. The economies of scale resulting from an amalgamation of these factors has made distributed

and parallel computing easier to access and more beneficial. The ease of access results not just from the availability of hardware but also a combination of other factors such as increased availability of software libraries and domain-specific expertise.

These developments at the high performance end of the computing platform spectrum have resulted in larger scale parallel computing systems, with the next generation of supercomputers capable of exa-scale computing. On the other end it has brought more and more computing closer to users, often termed “the edge”. This has increased interest in paradigms such as the Internet of things (IoT) and edge or fog computing. Apart from the opportunities to study, observe, and interact with the environment more closely with a faster turn-around time, computing at the edge can improve privacy by allowing users to better control computations and dissemination of data.

The availability of increased computing capability close to the edge through mobile computers has made it possible for simulations to be executed at the edge, enabling and improving a myriad of applications. Examples include automatic swarm surveillance and better adaptive systems using data driven simulations. Drone surveillance is being used in a multitude of applications ranging from vehicle tracking to monitoring wildfires, as presented in studies such as (Peng, Silic et al. 2015) and (Casbeer, Beard et al. 2005). For monitoring dynamic systems like these, it is important to be able to quickly and reliably predict the future state of the system to optimally reposition and reconfigure, sensors and resources. In case of the wild fires the predicted state could be the shape of the fire front and in the vehicle-tracking scenario, this could be the location of the tracked vehicle. Traditional methods employed a centralized

simulation based approach. Being able to compute these simulations close to the source of information eliminates the need to rely on instructions from a central location and/or the need to relay the sensor data to a central server. This not only allows these systems to adapt faster but also makes them more fault tolerant. Recognizing these advantages many such systems have been proposed in the literature. Works such as (Long and Hu 2017) and (Merino, Caballero et al. 2012) discuss cellular automata based simulations to predict the state of the wild fire and (Fujimoto, Guin et al. 2014) discusses a simulation based approach to track vehicles in urban environments. These approaches augment simulations with dynamic data from the environment to optimally reposition sensors, e.g. using live video to reposition/reconfigure a swarm of drones. Section 2.1 discusses a vehicle tracking scenario in greater detail. Such systems are broadly classified as dynamic data driven application systems (DDDAS) and will be discussed in greater detail in section 1.3.

These developments introduce a new set of challenges. In data centers power consumption is a concern because it forms the largest proportion of operating costs. Mobile systems are typically powered by batteries. This introduces constraints due to a limited energy budget.

While power and energy have been heavily studied in the embedded and mobile computing communities, and increasingly is being studied in high performance computing, very little attention to date has focused on understanding and developing techniques to minimize power and energy consumption in parallel and distributed discrete event simulations. This is the focus of the work described here.

The work described here begins with a characterization of the energy consumption behavior of parallel and distributed simulations. The insights gained from these studies motivate the development of algorithms for energy efficient parallel and distributed simulations and infrastructure to support their execution.

1.1 Background

We begin with a discussion of over-arching concepts that pervade this body of work. Specifically, hardware platforms and parallel and distributed discrete event simulation are discussed. Each of these is a large topic in itself, so the goal here is to limit the discussion to concepts needed to understand the research described later.

1.1.1 Hardware Platform: Parallel and Distributed Computing System

Parallelism in computing systems can be achieved at different levels of granularity, generally classified at bit level, instruction level, thread level and program level. This body of work and all the following discussion focus on the thread and program levels of parallelism in the context of simulation applications. A widely used classification of thread level parallelism is Flynn's taxonomy (Flynn 1972) that describes parallelism in terms of instructions and data streams. Single instruction stream with a single data stream or SISD forms the computing class for sequential computers. Two classes of interest in this work of study are the ***Single-Instruction Multiple-Data (SIMD)*** and ***Multiple-Instruction Multiple-Data (MIMD)***. SIMD processors execute the same stream of instructions. A fundamental characteristic of the SIMD processors is that they operate tightly synchronized with each other. First used in ILLIAC IV super computer, SIMD parallelism is now used in **Graphical Processing Units (GPUs).**

With the wider availability of commodity processors, supercomputers and in general parallel computing moved to MIMD. Unlike SIMD, MIMD processors compute asynchronously, i.e., each computing element (processor or thread) executes independent of each other. Hence external synchronization might be required for MIMD execution, for example to avoid race conditions and data hazards. An example of a MIMD system of increasing interest is a micro-cluster (Keville, Garg et al. 2012, Ou, Pang et al. 2012). Micro-clusters are energy-efficient high performance computing platforms that utilize commercial processors that were originally developed for cellular phones or tablets. Micro-clusters offer the potential to realize computation intensive applications on energy-constrained platforms such as drones or space vehicles.

Parallel computing systems can also be largely categorized into two classes based on the memory model employed. The first class is ***shared memory systems***. From an application or program perspective, all the processors in the system have access to a common memory system. This allows the processors to communicate by addressing a common location in memory, removing the need for redundant copies of data. A critical limitation of this class of systems concerns scalability arising from the fact that realizing fast access to globally shared memory is difficult. The second class is ***distributed memory systems***. Processors in these systems are associated with private memory and communicate with each other using messages sent over a communication network. There are other classes such as Non-uniform Memory Access (NUMA). It can be noted that distributed memory systems are the most general, in the sense that other memory models can be simulated on top of a distributed memory system. The work described here

primarily focuses on the distributed memory model, and hence the word distributed is sometimes omitted in the discussion.

The *distinction between parallel and distributed computing* is important. It should be noted that these have somewhat overlapping definitions in the literature, making a clear distinction difficult. But for this body of work, as is generally assumed, the distinction is made based on the tightness of coupling, which derives from communication delays. As a result, parallel computing systems often consist of a large number of less powerful processors that are tightly coupled with lower cost of communication, for example a GPU. Distributed computing systems on the other hand are composed more powerful processors that have a high cost of communication, e.g. networked high performance computing (HPC) clusters. These can be combined as in the case of a cluster of GPUs.

One final classification that is of interest here is the organization of the computing elements or the distributed computing architecture. Two major classes are the ***Peer-to-Peer (P2P)*** or de-centralized (also referred to as distributed) model and the ***Client-Server (CS)*** or centralized model. The behavior of clients in a CS system is governed by the server, which is generally assumed to be a node with access to larger amount of resources like memory and energy. In a P2P system, on the other hand, nodes collectively decide the behavior of system. Another notable consideration is that the clients in CS systems can communicate with each other only via the server, whereas the nodes in P2P systems can communicate directly with each other. It must be noted that the nodes in a P2P system and clients in a CS system can be heterogeneous in terms of the resource

constraints and functionality. Although it is easier to monitor, develop and maintain a CS system, P2P systems are generally comparatively more scalable and fault tolerant.

An important concept that relates to the performance of the parallel and distributed computation is speedup. Speedup is used to measure the performance gain of the parallel version of an algorithm/implementation over the sequential. It is defined as the ratio of the time required to execute the sequential algorithm and the parallel algorithm.

$$s(n) = \frac{T_{seq}}{T_{par}(n)} \quad (1)$$

Where,

s(n) is the speedup achieved by the parallel algorithm by using n processors.

T_{seq} is the time required for the execution of the best sequential algorithm.

$T_{par}(n)$ is the time required for executing the parallel algorithm on n processors.

It should be noted here that although it is always possible to convert a parallel algorithm to a sequential one by setting the number of processors to one, but T_{seq} might not be equal to $T_{par}(1)$ because the parallel implementation running on a single processor may not be as efficient as one designed only for sequential execution.

1.1.2 Discrete Event Computer Simulation

A dominating application of parallel and distributed computing, ***modeling and simulation (M&S)*** has been extensively utilized in fields ranging from the natural sciences and engineering to entertainment for some time. The history of M&S goes back to roman war-games. Modeling refers to the act of developing a representation for real or contrived events or system. A model is an abstraction of the system being investigated and is generally composed of models of constituent sub-systems. Simulation refers to the study of the behavior of a model over time. Here, we are concerned with ***computer simulation*** of models where the model is represented on a computer. Computer simulations are widely used when the system under investigation is too costly or time consuming, or even impossible to observe, or ethical constraints preclude experimentation. Computer simulations are widely used in many disciplines and enable to analyze "what if?" scenarios, assisting in analysis, decision-making and iterative enhancements.

Two classes of simulations are continuous and discrete event simulations. In a ***continuous simulation***, the state variables of the system are viewed as changing continuously over time. A simple example of a continuous model is plotting the trajectory of a projectile. In a Newtonian system, it is possible to exactly compute the position and velocity of the projectile for any given time instant. Unlike continuous simulations, in a ***discrete event simulations*** state changes are based on a sequence of events. The state of a simulation is depicted by a set of variables that completely describes the system at a given point in time (Sokolowski 2008). These variables are also known as the ***state variables***. An ***event*** as described in (Sokolowski 2008) is an instantaneous occurrence

that may cause a change in the state of the system. The events in computer simulation are generally formed by a tuple <*event_id, timestamp, event_data*>. The computer simulation program can be characterized as a set of event handler functions or simply ***event handlers*** that model a specific event and implement changes in state based on the *event_id*. The *timestamp* denotes the simulation time (defined shortly) when the simulation program processes the event. Here, we are concerned with discrete event simulations.

A fundamental concept in all simulations is time. Due to the interaction of different levels of abstractions and contexts, there are different notions of time:

- *Simulation time* is defined by (Fujimoto 2000) as "the totally ordered set of values where each value represents an instant of time in the physical system being modeled". That is the simulation time represents an abstraction of the time in physical system. The current simulation time forms part of the state of the simulation.
- *Wallclock time* is defined as the time corresponding to the real-world during and in context of the execution of the computer simulation program. This notion of time is used to compute speedup, for example. Wallclock time corresponds to the time represented by the hardware clock of the processor executing the simulation.

The discrete event simulations can be further classified based on how simulation time advances during the simulation. In a ***time-stepped simulation***, time advances in fixed increments of simulation time. In an ***event-driven simulation***, simulation time advances based on the timestamp of each event. It can be noted the time-stepped

simulation can simulated using an event-driven paradigm. Hence for the most part, simulation programs discussed in this work are event-driven, unless otherwise noted.

Finally, based on the relation between the simulation time and wallclock time during the execution of the simulation program, the simulations can be classified as a *real-time* or an *as-fast-as possible* simulation. In ***real-time simulations*** simulation time advances synchronously with wallclock time. This can be important when the simulation interacts with humans (human-in-the-loop simulations), e.g. video games, or where the simulation interacts with real hardware devices that form part of the simulation system (hardware-in-the-loop simulations), e.g. simulations used to test hardware devices. Another class of this type of simulation is scaled real-time simulation. In this case simulation time advances are proportional to advances in wallclock time. For real-time simulations this proportionality constant is 1. In ***as-fast-as-possible simulations*** simulation time advances do not have any direct relationship with wallclock time. The goal of these simulations is to produce simulation results as quickly as possible, hence the name. Scientific computing simulations generally fall under this category. Although most of the examples presented in this work discuss as-fast-as-possible simulations, the concepts and ideas presented can be easily applied to real-time simulations.

1.1.3 Parallel and Distributed Discrete Event Simulation

Parallel and distributed simulations (PADS) attempt to speed up the execution of the simulation through parallel execution. PADS can be defined as the technologies that enable the execution of computer simulations on parallel and/or distributed computing systems.

The benefits of parallel and distributed computing translate to the five prime benefits (Fujimoto 2000) of PADS over its sequential counter-part. As mentioned earlier the parallel and distributed execution results in a faster execution time. Given a constant amount of time budgeted for simulation this can result in an increase in the fidelity of simulations or reliability of stochastic simulation[1] results or both. Secondly, the distribution of simulation provides the possibility of geographically distributing the simulation. One example is where agents interacting with the simulation are geographically separated, for example online video games (virtual environment simulation) over the Internet. A second example is the case where the simulation components are not co-located, for example hardware-in-loop simulation with geographically separated hardware components. Distributed simulation can foster re-use of simulations by allowing the use of a middleware (or Runtime Infrastructures) to integrate separately developed simulators. Also called federating simulations, this is can be less costly compared to developing new simulations. Distribution of simulations can improve fault tolerance by reducing dependence on a single computing system. Finally, for the resource-constrained platforms, resource sharing is a valuable consideration that arises due to the heterogeneity of the computing elements engaged in the simulation.

The parallelization of the simulation is generally achieved by dividing the model into quasi-independent parts, called ***Logical Processes (LPs)***. A fundamental characteristic of PADS is that the LPs do not share any state variables (even in shared-memory systems) and hence have independent state vectors. Parallelization is achieved by assigning these logical processes to individual processes in a parallel or distributed

[1] Simulations involving random variables with associated probabilities are called stochastic simulation.

[2] Multiple lookahead optimization and specializations are possible. Lookahead can be

computing system. Two methods to generate quasi-independent parts are to divide the simulation space and the other is to divide the simulation time. The level of independence of LPs is a key factor that determines the parallelism that can be obtained. The greater the amount of interactions among the LPs, the more communication and synchronization will be required.

Time parallel simulations use an approach where simulation time is partitioned into time segments, and an LP assigned to simulation each one. They provide the opportunity for massive amounts of parallelism and may require less communication across LPs. The main challenge concerns dependencies that cross time-interval boundaries. A state matching process is required to address this problem. State matching typically depends on the model, and simulation technique and can be time consuming. Simulation computations parallelized using parallel prefix computations fall under this category of parallelization.

As pointed out earlier, models of systems can usually be broken down into constituent components. The ***spatial decomposition*** is defined by modeling each of these components using a LP. This is the approach that is generally used for implementation of PADS.

A central objective of PADS is to maintain the correctness of the simulation, that is, to ensure the result of a PADS program should match that of the sequential version. More specifically, all the LPs must process the events in strictly non-decreasing time stamp order for the simulation to be correct. This is referred in the literature as the ***causality constraint*** (Fujimoto 1990). Although this can be easily achieved in a

sequential simulation by using a central list of events, this is not trivial for PADS. For PADS, the arrival of events at LPs need not be in time stamp order. To ensure that the events within each LP are processed in correct order a ***synchronization algorithm*** is needed.

The synchronization techniques for parallel and distributed discrete event simulations can be broadly classified into two major categories, termed conservative and optimistic approaches. Apart from the specific constraints, as will be pointed out shortly, the most notable difference among these approaches is that the conservative approaches prevent any violation of the causality constraint, whereas the optimistic approach allows a temporary violation which is recovered at a later stage during the course of simulation.

Conservative synchronization approaches can be traced back to what is now known as the null message algorithm or the ***Chandy-Misra-Bryant (CMB) algorithm***. This algorithm was presented in two independent works (Chandy and Misra 1979) and (Bryant 1977), and is perhaps the most well-known conservation synchronization approach. The core of this approach lies in deadlock avoidance. The CMB algorithm was initially developed for analyzing queueing network simulation with applications in communication networks and hardware logic simulations. This is reflected in the assumptions that govern the application of the CMB synchronization. CMB assumes that the communication channels among the LPs are ***static*** and that the LPs guarantee that messages are sent in time-stamped order. Furthermore, the communication channels are assumed to be FIFO (order preserving) and guarantee delivery. A major implication of these assumptions is the LPs as well as communication channels cannot be added or removed dynamically during the simulation.

Each LP includes *O* outgoing channels and *I* incoming channels. Each incoming channel in the LP has a FIFO queue associated with it. The LP chooses the smallest event among queues associated with all the incoming channels. The LP processes the event if it is safe to do so. An LP considers an event to be ***safe*** if it can guarantee that processing the event will not violate the causality constraint.

LPs synchronized by CMB avoid deadlocks by informing its neighbors the smallest possible timestamp of an event it can later send over a communication channel. The LPs communicate these using messages, known as ***null-messages*** (hence the name null-message algorithm). Each LP sends out null-messages to all its outgoing channels after it processes an event. Combined with the assumptions of the system, this implies a guarantee that the time stamp of the Null Message, T_i, is a lower bound on the timestamp of any event that can be received on the channel *i*. An LP deems an event safe if the timestamp of the event, *t*, is smaller than the minimum of the lower bounds of the timestamps on all incoming channels. Succinctly put, for event e in LP *l*, with a timestamp *t.*

$$safety(e) = \begin{cases} True, & t < T_i \ \forall \ i \in I \\ False, & otherwise \end{cases} \quad (2)$$

Where *I* represent the set of all incoming channels of LP *l*.

In CMB the number of null-messages exchanged in the system can be large, representing a significant overhead. An alternative approach to reduce the number of null messages is to exchange null-messages only when an LP determines a possible deadlock (Silberschatz, Galvin et al. 2006).

An LP determines the timestamp of null messages based on a combination of two pieces of information. The first piece is inherent to the LP's simulation state, for example the current local time of the LP or the minimum of the lower bounds on all its incoming channels i.e. $\min(T_i)\ \forall\ i \in I$ or a combination of these. The second piece of information is generally derived from the knowledge of the system being modeled and is referred to as the lookahead. **Lookahead** can be defined as the difference between the timestamp of the earliest event the LP can create[2] and the current local simulation time of the LP. In essence, the lookahead can be viewed as the simulation time in which the LP will not affect the state of any other LP. For example, consider a communication network simulation with nodes *A* and *B*. The lookahead for Node *A* can be set to the time required for a packet of data to travel from *A* to *B*. As can be noted, in general depending on the application the lookahead can vary during the course of a simulation. It is useful to note here that the performance of conservative synchronization is tied to the quality and computability of lookahead and is generally acknowledged that a system with a good lookahead can perform well.

The CMB algorithm is an asynchronous algorithm. Its counterpart, synchronous algorithms utilize global synchronization points. The computation executes through a sequence of epochs (or barriers, as is known in the parallel and distributing computing literature) where each involves determining those events that can be executed in this epoch with timestamps that are guaranteed to be smaller than any event that might later be received. Well known synchronous algorithms include **YAWNS** (Nicol 1993) and

[2] Multiple lookahead optimization and specializations are possible. Lookahead can be specialized for each channel, referred to as Link lookahead, or for an entire LP (as is presented here) that is LP lookahead.

Bounded Lag (Lubachevsky 1989). The synchronous algorithms can be adapted to peer-to-peer or client-server based systems, depending on how the barrier is implemented. For example, a centralized barrier implementation can be used for a client-server based implementation of synchronous algorithms, whereas tree barriers or butterfly barrier implementations can be used for a peer-to-peer based system.

The other class of synchronization algorithms is termed ***optimistic***. As pointed out earlier, optimistic synchronization algorithms allow LPs to violate the causality constraint and recovers from violations by rolling back if a violation is detected at runtime. The inherent goal of optimistic synchronization is to synchronize only when required, hence possibly exposing better parallelism. The origins of this class can be tied to the ***Time Warp algorithm*** (Jefferson and Sowizral 1982, Jefferson 1985). Apart from the general paradigm of LPs exchanging time stamped messages/event, the algorithm makes one other assumption about the underlying communication channels. The assumption is that the channels guarantee the delivery of messages. The key property that relaxes the other assumptions of the conservative synchronization algorithms is that out-of-order execution is allowed and although expensive[3] the system can recover from it. This translates to the possibility of dynamic creation and removal of LPs, change in network topology, and out-of-order message delivery by communication channel during the simulation[4].

In Time Warp LPs process events as they become available. If a message/event arrives with a time stamp less than the current local simulation time of an LP, the LP rolls back to the time stamp of the received event.

[3] The goal being, savings due to resulting parallelism outweighs the cost of recovery.
[4] It should be noted that conservative algorithms like YAWNS could do these as well.

Time Warp, and in general optimistic synchronization algorithms provide an application independent algorithm that can achieve good performance. This not only allows a general-purpose synchronization algorithm (indicative of availability standardized libraries), but also makes it easier to federate different simulations. But the resource efficiency and overhead associated with the optimistic synchronization algorithms is highly dependent on the overhead and resource efficiency of mechanisms responsible for rollback, e.g. state saving and computing global virtual time. In addition, it might not be always possible to do an accurate roll back to a state in simulation (Nicol and Liu 1997) resulting in inaccurate results or inconsistent repetitions.

1.2 Energy and Power of Computation is a Concern

As touched upon earlier, power consumption has become a major concern for many parallel and mobile computing applications. The increased demand for energy required by mobile computing devices, mostly fuelled by the exponential increase in the computational power and capabilities, has far outpaced battery technology (Want 2014). The need to reduce energy use is clear in mobile and embedded computing where reductions result in increased battery life or enable the use of smaller batteries thereby reducing the size and weight of devices. In high-end computing, energy consumption is a dominant cost associated with operating large data centers and supercomputers, and a substantial amount of effort has gone into developing techniques to mitigate this expense. Power consumption has become the key factor preventing substantial further improvements in clock speed and now limits computer performance. It has been cited as a major obstacle to creating supercomputers yielding exascale performance (Ang, Bergman et al. 2014). The U.S. Department of Energy has specified 20 megawatts as the

goal for the maximum power consumption for such a machine. Power is a major expense in data centers used for cloud computing. Data centers were estimated to consume approximately 70 billion kW-hours or 1.8% of the total electricity consumption in the U.S. in 2014 (Shehabi, Smith et al. 2016).

Energy is the capacity of a system to perform work. It is typically measured in units called joules where one joule is the work performed by an electrical circuit to move a charge of one coulomb through an electrical potential difference of one volt. Power is the amount of energy consumed per unit time with one watt of power defined as the expenditure of one joule of energy per second.

Minimizing energy usage and power consumption are not the same thing (Unsal 2008). For example, decreasing the clock rate of the processor can lead to less power consumption. However, this will usually lead to longer execution times and an increase in the total amount of energy needed to complete a given computation. Battery operated devices are *energy-constrained* systems because they operate with a finite amount of available energy; thus, a design goal might be to minimize the amount of energy utilized by the computation, subject to certain execution time constraints. On the other hand, in *power-constrained* systems such as supercomputers and data centers the amount of available energy is effectively unlimited, but a design goal may be to minimize the amount of time required to complete the computation given a certain maximum level of power consumption, or to minimize peak power consumption, subject to certain execution time constraints.

Power-aware and *energy-aware* systems are those where power or energy consumption is a principal design consideration. For example, power-aware systems may utilize techniques to change the system's behavior based on the amount of power being consumed. Energy-aware systems may modify the operation of the system based the amount of energy remaining in batteries.

It should be noted that minimizing execution time does not necessarily result in minimal energy consumption. Energy consumption is affected by many factors, e.g., the operation of the memory system, the number and complexity of computations performed by arithmetic circuits, and importantly, the amount of inter-processor communication that is required. A parallel or distributed computation that executes in a shorter amount of time may consume more energy and more power if more communications are required.

More specific related works from literature are discussed in the chapters that follow.

1.3 DDDAS

Trends such as ubiquitous computing, edge computing and the Internet of Things highlight that an area of increasing interest concerns the execution of distributed simulation programs on mobile and embedded computing platforms. One specific example of a class of applications where energy consumption is an important concern are dynamic data driven application systems (DDDAS) (Darema 2004). These are applications that interact closely with real-world systems typically with the goal of optimizing the system or to adapt the monitoring subsystem. DDDAS applications typically operate in a control loop that involves

(1) Collecting and processing data extracted from sensors and other devices,

(2) Executing algorithms or simulations utilizing these data to inform decision making processes, and

(3) Reconfiguring the system to optimize it along some dimension or to adapt the monitoring process.

For example, in transportation system applications simulations may be used to predict future system states in order to guide mechanisms to manage congestion or adapt system monitoring. Because DDDAS applications interact directly with operational systems, it may be advantageous to embed computational devices within the system being managed, e.g., a sensor network. Some real-world examples of DDDASs' using mobile computing platforms are presented in (Fujimoto, Hunter et al. 2007, Kamrani and Ayani 2007, Madey, Blake et al. 2012). In this work, our particular interest could be tied to DDDAS systems where distributed simulations executing on mobile computing devices are used in the DDDAS control loop. For example, distributed simulations may be embedded within a mobile sensor network monitoring a forest fire or vehicular traffic in order to relocate sensors to track the evolving system or to concentrate monitoring activities in areas of particular interest.

In the computing industry, Microsoft's public experiment (Microsoft-Edge 2016) comparing the battery life of four identical laptops running different web browsers demonstrates that energy consumption of software is now an important performance consideration for software in industry along with execution time, which previously had been synonymous with performance of software and algorithms.

1.4 Research Contributions

The contributions of this body of work can be broadly classified in two major categories. The first focuses on developing an understanding of the energy and power consumption characteristics of distributed simulations. The second focuses on using this understanding to develop new algorithms and infrastructure to support energy efficient execution of parallel and distributed simulations.

1.4.1 Understanding Energy and Power Consumption Characteristics of Distributed Simulations

Specific contributions include:

- **An empirical study of energy and power consumption in distributed simulations.** Through empirical measurements of conservative synchronization algorithms (Biswas and Fujimoto 2017), it is demonstrated that distributed execution incurs a significant overhead in energy consumption, and different synchronization algorithms demonstrate different behaviors. Architectural choices significantly impact the amount of energy and power that is consumed. Performance metrics are proposed. The study was the first of its kind in the parallel and distributed simulations community (Fujimoto and Biswas 2015).
- **Techniques to create energy profiles of distributed simulation to quantify the energy consumed by its various aspects** (Biswas and Fujimoto 2016). Creating energy profiles is not straightforward because high precision, low overhead energy measurement mechanisms may not be available, and it is not straightforward to determine the amount of energy consumed by different

components of the distributed simulation. The proposed model differentiates the energy consumed by the functional components of distributed simulation, that is energy consumed by the distributed simulation engine versus simulation application code, and energy consumed for computation versus that required for communication. Empirical data are presented to validate the energy profile that is obtained. A study is described to illustrate this methodology to profile a distributed simulation synchronized by the Chandy/Misra/Bryant synchronization algorithm for a queuing network simulation. Results of the study corroborated the significant energy overhead of distributed execution. More specifically, the distributed simulation engine and communication functions were found to consume a significant portion of the total energy for the benchmark application.

- **Empirical studies to quantify the possible improvements in energy and power consumption of distributed simulations, showed a significant reduction can be achieved by optimizing communications and CPU power management.** Message aggregation resulted in as much as 7.5 times reduction in the average joules required per byte (Fujimoto, Hunter et al. 2017).

These measurements suggest that techniques to optimize the energy costs associated with the distributed execution of simulations may yield significant benefits. More specifically, these measurements indicate that the development of new approaches for synchronization may yield significant improvements in energy overhead.

1.4.2 Optimizing Energy and Power

The second set of contributions includes proposals for algorithmic and infrastructural approaches to effectively manage and reduce energy consumption.

- **A lower bound on the energy required for distributed simulations is characterized and Zero-Energy synchronization algorithm is defined. An oracle based synchronization scheme is proposed to achieve zero-energy synchronization for arbitrary distributed simulation code** (Biswas and Fujimoto 2018). Zero energy synchronization is proposed as goal for distributed simulation synchronization algorithms and implementations. This concept is based on the observation that synchronization does not directly contribute to the computational results produced by a distributed simulation but can represent a significant source of energy consumption in distributed simulations.
- **Low Energy YAWNS (LEY), is proposed as a practical approach to reduce the energy consumed for synchronization** (Biswas and Fujimoto 2018). It is shown that LEY can, in principle, achieve zero energy synchronization for a large class of distributed simulation applications. The energy consumption of distributed simulations is modeled to analyze and compare the energy consumption of various synchronization algorithms. In addition to this analysis, an empirical study demonstrates that LEY consumes much less energy than a conventional implementation of YAWNS that does not attempt to minimize energy consumption, and LEY achieves energy performance approaching zero-energy synchronization for a set of benchmark applications.

- **Green Runtime Infrastructure (G-RTI) middleware to improve energy efficiency for DDDAS applications on heterogeneous platforms, and illustrated for a connected vehicle application** (Biswas, Hunter et al. 2018). Middleware is required to support and interface multi-modal DDDAS with back-end and other computing facilities. Middleware is also needed to support distributed simulations and emulations needed in earlier phases of system development. Developed to serve as a platform for research in energy reduction techniques for middleware services, G-RTI provides a common platform for simulation, emulation and deployment of such systems. Major design goals include support of a wide variety of devices and application contexts, reduced development effort, providing a platform for energy efficient deployment and the possibility of thin-as-required clients. Benchmarking studies show that G-RTI services performs better than comparable middleware implementations. In addition, its application is demonstrated through a use case for an end-to-end implementation of a connected vehicle application.
- **A linear recurrence for simulating G/G/1 queues in parallel is proposed and empirically analyzed.** Queues have been widely used to model and study the behavior of physical systems. The proposed recurrence can be used to simulate a wide variety of global first-come-first-serve (FCFS) based queueing networks. These recurrences are composed of simple and widely used parallel processing operations. This eases the development of the queueing network simulation and also allows the simulations to be implemented using parallel processing primitives available in most parallel processing frameworks. As more and more specialized

hardware become available, use of the proposed recurrence would allow further improvement in energy and computational efficiency of queueing network simulation.

- **Composition of parallel prefix is proposed for composing a sequence of parallel prefix scans.** Prefix scan is extensively used for applications in science and engineering, making parallel prefix scan an important parallel primitive. It is common for applications to stack prefix scans in a series of parallel prefix scans, with results of one feeding the next. The composition of such a series of parallel prefix computations is described. The composite parallel prefix computation is demonstrated to be more efficient in terms of time and energy consumption using an empirical analysis of queueing network simulations. Theoretical analysis points out that the composite implementation of the recurrences is k times faster than corresponding parallel implementation of the recurrences using k parallel prefix scans. This is supported by empirical experiments conducted using simulation of a 2D toroidal queueing network. Also, the composed implementation is as much as 450 times faster than sequential implementation. It is also up to 30% faster and at an average 13% more energy efficient than a parallel implementation of the recurrences using the fastest library primitives available.

1.5 Organization

The organization of this document follows that of the previous section. The initial chapters build the required understanding and insight into power and energy consumption behavior of distributed simulations. Chapter 2 commences by discussing an example to

motivate and concretize the importance of energy for PADS. This is followed by a discussion of metrics to study energy and power consumption of PADS. The chapter concludes with empirical studies to characterize the energy consumption of PADS with different computing architectures. Chapter 3 continues this work and describes an energy model for distributed simulations and discusses techniques to empirically construct energy profile for distributed simulations using the model. It closes with benchmarking experiments that provide insight into the energy consumption of different aspects of distributed simulation and validates the proposed empirical techniques.

The next chapters build upon the understanding and trends outlined by the initial chapters. Chapter 4 discusses and describes G-RTI middleware. The chapter then outlines energy and performance benchmarks for G-RTI and its services. Finally, the chapter closes with a case study of a DDDAS using G-RTI. Chapter 5 introduces the concept of zero-energy synchronization, proposes an oracle-based approach to achieve zero-energy synchronization and discusses the development of LEY. The chapter concludes with empirical analysis comparing the LEY with YAWNS and a zero energy synchronization algorithm. Chapter 6 shifts the focus from general purpose schemes to improve the efficiency of distributed simulations to application specific approaches. The chapter presents an exploration for simulation of FCFS based G/G/1 queueing network simulations. It also presents a discussion and analysis of composition of parallel prefix scans. The chapter then closes with an empirical analysis and discussion of a 2D toroidal queueing network simulation on CPU based and GPU based computing systems.

2 DISTRIBUTED SIMULATION AND ENERGY

Power- and energy-aware computing is increasing in importance for parallel and distributed simulation systems and applications. The main contribution of this chapter is to highlight the increasing importance of power and energy consumption in parallel and distributed simulations. We propose performance metrics to quantitatively assess energy consumption in parallel and distributed simulations and report initial empirical measurements of the energy consumed by conservative synchronization algorithms.

The next section describes a motivating application to highlight the relevance of energy consumption in distributed simulation. This is followed by a discussion of related work in this area by briefly surveying power- and energy-aware computing. Metrics for measuring energy and power consumption in distributed simulations are then proposed. The two distributed simulation systems examined in this study – a peer-to-peer system using the Chandy/Misra/Bryant algorithm and a client-server architecture using the YAWNS algorithm are then described. The experimental configuration and benchmark programs used in this study are presented, followed by a discussion of the power measurement methodology that is used. Experimental results are then presented and discussed, followed by concluding remarks and discussion of areas requiring further investigation.

2.1 A Motivating Application

Embedded mobile distributed simulations can be used to create adaptive sensor networks to monitor dynamically changing physical systems. For example, consider a

collection of small, battery-operated unmanned aerial vehicles (UAVs) tasked with monitoring a physical system, e.g., tracking a set of vehicles moving throughout a city, monitoring the spread of a forest fire, or assessing the dispersion of a hazardous chemical plume following an accident (see Figure 2-1). Assume each UAV is equipped with sensors, an on-board computer, and wireless communications. Each UAV may initially be assigned to monitor a certain geographical area. It collects information concerning the state of the physical system in its immediate vicinity. Collectively the team of UAVs may then execute a distributed simulation to project the future state of the system, e.g., to project the location of the fire some time into the future in order to determine how best to relocate the UAVs in order to continue monitoring its spread. In some cases, more UAVs may be assigned to monitor regions of interest, e.g., areas with higher traffic congestion, a larger density of fires, or higher concentrations of chemicals, leaving fewer UAVs to monitor areas projected to be less important to the monitoring activity. One can envision other similar surveillance applications involving teams of people carrying handheld devices or autonomous battery-powered ground vehicles.

Adaptive sensor networks such as these could utilize centralized computing capabilities where sensors report information back to a command center where predictive simulations could be executed and the network reconfigured accordingly. Placing the simulations within the sensor network, offers several advantages. First, it reduces or eliminates reliance on connectivity to the central command center, mitigating a potential point of failure. Further, a distributed implementation enables greater scalability than the centralized approach; one can envision many teams of UAVs that collaborate to monitor larger scale systems than would otherwise be possible. In certain applications embedding

the simulation within the physical system itself enables faster response time for latency-critical applications.

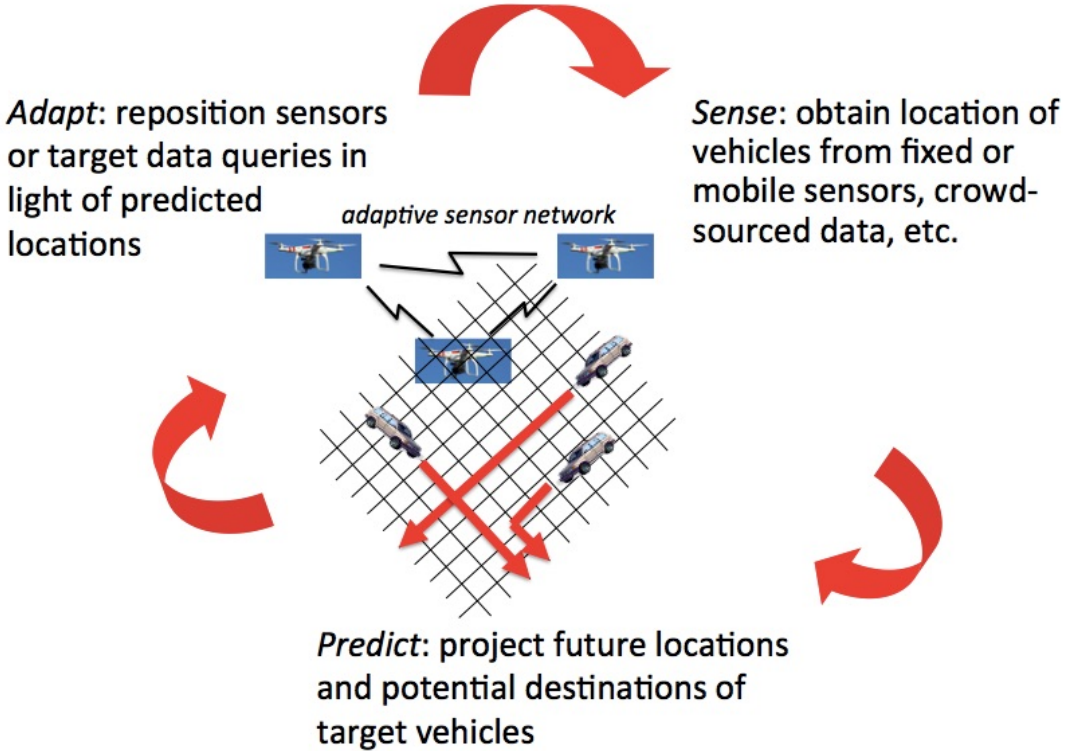

Figure 2-1: Notional Diagram of a mobile data-driven distributed simulation system for monitoring traffic

Systems such as these are an example of DDDAS, presented in section 1.2. DDDAS involves incorporating live data from instrumented systems into executing applications in order to optimize the system and/or steer the measurement process. DDDAS has been studied in a variety of applications. Our focus here is concerned with using embedded distributed simulation to implement the second step of the DDDAS processing cycle, i.e. predict future system states. It is clear that for these situations, energy consumption is an important concern when the simulation operates within battery-

operated mobile devices. Exploitation of the DDDAS paradigm in UAVs is an active field of study and are discussed in (Kamrani and Ayani 2007, Madey, Blake et al. 2012).

2.2 Related Work

There is a substantial literature in power- and energy-aware computing systems, and a variety of techniques may be employed. Dynamic voltage and frequency scaling (DVFS) is concerned with altering the voltage and/or clock speed of the processor by taking into consideration energy and performance constraints (Hua and Qu 2003, Ge, Feng et al. 2005, Freeh, Lowenthal et al. 2007). Dynamic power consumption in CMOS circuits is proportional to FV^2 where F is the frequency at which the circuit is clocked, and V is the power supply voltage. Several commercial microprocessors support modification of the processor's frequency and voltage to trade off power consumption and performance. Scheduling algorithms for embedded systems have been designed to balance energy saving with meeting real-time deadlines, e.g., see (Quan and Hu 2001, Saewong and Rajkumar 2003, Cho, Liang et al. 2011). Processors also commonly provide different modes of operation that utilize different amounts of power. Some work examines the utilization of these modes of operation, sometimes in conjunction with DVFS (Niu and Quan 2004, Hoeller, Wanner et al. 2006, Bhatti, Belleudy et al. 2010). Other research examines issues such as predictive modeling of energy and power (Williams, Waterman et al. 2009, Keckler, Dally et al. 2011, Czechowski and Vuduc 2013) and embedded systems evaluations (Rajovic, Carpenter et al. 2013, Rajovic, Rico et al. 2013, Stanisic, Videau et al. 2013, Grasso, Radojkovic et al. 2014).

To date, work in power and energy aware computing has largely focused on low-level aspects of the computing system. Existing work focuses on effectively utilizing specific hardware capabilities, the development of operating systems and compilers, and communication protocols (e.g., routing algorithms in sensor and ad hoc networks) to reduce energy usage.

Only a modest amount of work to date has addressed energy and power consumption in parallel and distributed simulations. Some work has focused on characterizing power consumption for scientific computing applications (Feng, Ge et al. 2005, Ge, Feng et al. 2010, Dongarra, Ltaief et al. 2012, Esmaeilzadeh, Cao et al. 2012). One early effort examined power consumption for disseminating state information in distributed virtual environments, highlighting dead-reckoning algorithms and tradeoffs between state consistency and power consumption (Shi, Perumalla et al. 2003). More recent work examined power consumption related to the implementation of data distributed management (DDM) services defined in the High Level Architecture (Neal, Kanitkar et al. 2014). Only a couple studies (Child and Wilsey 2012, Maqbool, Naqvi et al. 2017) have examined power and energy consumption of synchronization algorithms for parallel and distributed simulations, the primary focus of the work described here.

2.3 Energy and Power Metrics

Three key metrics of great interest are the amount of energy, power, and time required to complete a computation. These metrics may be traded off against each other. For example, as discussed earlier one can trivially reduce power consumption by reducing clock frequency at the expense of increased execution time. In high

performance computing contexts both execution time and power consumption are of interest, and balanced based on constraints such as a maximum level (ceiling) of power consumption. Similarly, in real-time applications both energy consumption and maximum execution time are important for best use of system resources while still meeting real-time constraints. For this reason, the product of energy (or power) and execution time, referred to as the energy-delay product, is sometimes used as a metric that simultaneously considers both energy consumption and execution time (Gonzalez and Horowitz 1996).

A central concern here is the amount of additional energy consumed by the parallel/distributed execution that takes into account parallel/distributed computing overheads such as interprocessor message communication and synchronization. For this purpose, we define a metric termed the *energy overhead*. Energy overhead refers to the amount of *additional* energy that is expended in the execution of a particular implementation of a parallel or distributed simulation on some hardware configuration relative to an energy-efficient sequential execution of the same computation. In some cases, it may be desirable to specify E_P (or $E_{P'}$) relative to the amount of energy consumed by the sequential execution by dividing E_P by E_s (or $E_{s'}$).

This definition is motivated by the traditional definition of speedup. Like the speedup definition, "performance" is defined relative to an efficient sequential implementation. Also like speedup, this definition is meant to encourage the development of approaches to minimizing energy consumption recognizing that a baseline amount of energy must necessarily be consumed by the computation, just as speedup recognizes that a certain amount of time is required to complete the computation on a sequential

machine. Energy overhead highlights the cost of the parallel/distributed implementation in terms of energy consumption.

Just as speedup may be determined using *strong* or *weak scaling*, similar methodologies apply in measuring energy overhead. In strong scaling the speedup is computed by comparing the execution time of a fixed sized sequential computation with a parallel implementation of the same computation distributed across a parallel processor. In this light *strongly scaled energy overhead* is computed as $E_P(N) - E_S$ where E_S is the energy consumed by a sequential implementation of the computation, and $E_P(N)$ is the energy consumed by a parallel/distributed implementation of the same computation distributed over N processors.

Alternatively, speedup computed using weak scaling involves scaling the size of the computation in proportion with the number of processors. Here, *weakly scaled energy overhead* is defined as $E_{P'}(N) - E_{S'}$ where $E_{S'}$ is the energy consumed by a computation C on a sequential machine and $E_{P'}(N)$ is the energy consumed by the same computation C executing on a single processor of the parallel/distributed machine with N processors and where the entire computation executed on the parallel/distributed machine is of size $C*N$. Weakly scaled energy overhead provides insight into the amount of additional energy consumed by the parallel or distributed computation as it is scaled to larger sizes in proportion to the size of the parallel/distributed computer.

The energy overhead is impacted by several factors. It clearly depends on the hardware configuration, including consideration of memory and communications circuits, and system software on which the parallel/distributed simulation executes. Energy

consumption depends on any energy-saving techniques such as DVFS that are used. Our particular concern is the overhead associated with the synchronization algorithm. As will be discussed next, energy consumption depends on the software architecture used for the implementation.

The above discussion focused on energy overhead. Similar metrics for *power* overhead can also be defined. In this context, the power overhead refers to the additional amount of power required to execute the parallel/distributed simulation relative to the sequential simulation.

2.4 Distributed Simulation Systems

In this study we compare two distributed simulation middleware approaches using conservative synchronization algorithms. These two approaches utilize a peer-to-peer and a client-server architecture, respectively. The context in which we envision these architectures to be deployed might be an embedded DDDAS application where the distributed simulation executes on a power-constrained mobile computing platform, possibly connected via wireless links to a local server. This architecture places the simulations in close physical proximity to online sources of data.

The "classic" approach to implementing a parallel discrete event simulation (PDES) program is to use a *peer-to-peer architecture* where each processor or node has approximately the same computational capabilities as other nodes. The logical processes (LPs) making up the PDES program are mapped to different computation nodes using a mapping algorithm or heuristic. LPs communicate directly with other LPs by sending messages to the appropriate nodes. Early work in PDES focused almost exclusively on

this approach, and to this day, this approach is often used to implement parallel or distributed simulation systems. This architecture is depicted in Figure 2-2(a) below.

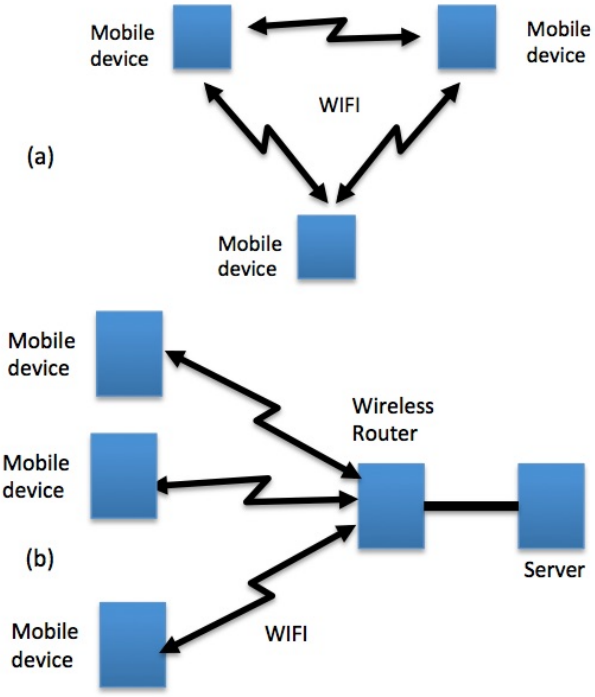

Figure 2-2: Peer-to-peer and client-server approaches

In the peer-to-peer architecture a distributed algorithm is typically used to implement synchronization. Asynchronous algorithms use direct peer-to-peer communications between LPs (or processors). The Chandy/Misra/Bryant (CMB) algorithm is perhaps the most well-known example of this approach (Bryant 1977, Chandy and Misra 1979). The principal source of energy overhead for this algorithm results from transmitting null messages between processors.

A second approach to distributed simulation is the client/server architecture, as shown in Figure 2-2(b). Logical processes execute within client processors while the simulation engine executes within the server. Here, in the context of the applications

described earlier, we envision a mobile server that might reside in a special device, e.g., a larger gasoline-powered UAV. In general, however, the server might utilize the same mobile processor as client nodes in which case the distinction between clients and servers is largely logical, or the server might utilize a different, likely more powerful machine.

Clients only communicate with the server; direct client-to-client communications are not allowed. Two key functions performed by the server include forwarding messages sent between LPs residing in different clients and synchronization among the LPs/clients. This architecture is sometime used in federated distributed simulations, e.g., those based on the High Level Architecture standard, where RTI services are for the most part implemented within the server.

A natural, straightforward approach to implementing synchronization in client-server architectures is using synchronous algorithms that utilize global synchronization points. The computation executes through a sequence of epochs where each involves determining those events that can be executed in this epoch with timestamps that are guaranteed to be smaller than any event that might later be received. Well known synchronous algorithms include YAWNS (Nicol 1993) and Bounded Lag (Lubachevsky 1989). A principal source of energy overhead for this algorithm lies in the barrier synchronization mechanism and lower-bound-on-timestamp (LBTS) computation that is required. Each epoch includes a barrier and computation of the LBTS value indicating the minimum timestamp of any event that might be generated in the future. More precisely, each client processor computes the smallest time stamp for any new message it might produce in the absence of receiving any additional messages as $T_i + L$, where T_i is the timestamp of the next unprocessed local event within the processor and L is the

lookahead for the processor. LBTS is defined as the minimum among all of these values produced by the different processors. All events with timestamp less than LBTS are safe to process.

2.5 Experimental Configuration and Benchmark Application

The experimental configuration is intended to mimic an embedded distributed simulation application where the distributed simulation executes within a set of mobile processors. In the peer-to-peer system the mobiles make up the entire hardware platform. In the client-server architecture we posit the server resides in a location where a power source is readily available so energy use within the server is of secondary importance.

Experiments were performed to compare the CMB (peer-to-peer) and YAWNS (client-server) algorithms. A LG Nexus 5 cellular phone with a quad-core Qualcomm MSM8974 Snapdragon 800 processor, 2 GB memory, and 16 GB storage was used as the mobile computing platform. While the systems we envision may not necessarily use cellular phones as their compute engine, they most likely will utilize the same mobile processors that are used in cellular phones. The phone runs the Android version 5.0.1 (Android Lollipop) operating system and was used in the peer-to-peer experiments. The same phone was used as the client in the client-server architecture. A laptop computer was used as the server for the latter architecture. All the experiments used two mobile devices. Hardware-based techniques to reduce power consumption such as voltage or frequency scaling were not used in these experiments.

All inter-processor communications utilize wireless links. In both cases the device's 802.11n Wi-Fi network interface was used for communications between

processors. A private wireless network was established among the devices to avoid interference resulting from Internet traffic. The cellular network capability of the phone is not used in these experiments.

The energy and power consumption data are derived from direct measurement of the Android device. Specifically, the value of the instantaneous power being consumed by the device is calculated by multiplying the instantaneous battery current given by the constant integer BATTERY_PROPERTY_CURRENT_NOW and the current battery voltage level given by the constant string EXTRA_VOLTAGE. Both of these appear in the "BatteryManager" class of the "android.os" API. The instantaneous power consumption so obtained is then used to calculate the energy consumption over time.

2.5.1 Energy Used by Synchronization Algorithms

An initial set of experiments was conducted to measure the amount of power used by the synchronization algorithm. This was accomplished by creating benchmark programs where each LP only processed local events, and did not exchange events with other processors. This ensures that interprocessor communication is only utilized by the synchronization algorithm rather than passing event messages. In these experiments each LP is initialized with some number of local events, and processing each event causes one new locally scheduled event to be scheduled with a fixed time stamp increment. A set of experiments was then performed using the CMB and YAWNS implementations as lookahead was varied. Because no events are scheduled between processors, the lookahead could be set to arbitrary values without concern of violating lookahead

constraints. In these experiments the simulation benchmark program is the same across all lookahead values and both synchronization algorithms, enabling fair comparisons.

The amount of energy consumed by the benchmark programs for different lookahead values are shown in Figure 2-3. In this figure both lookahead and energy are plotted on logarithmic scales. It is seen that lookahead can have a dramatic effect on the amount of power consumed by the synchronization algorithm – two orders of magnitude across the lookahead values used in these experiments.

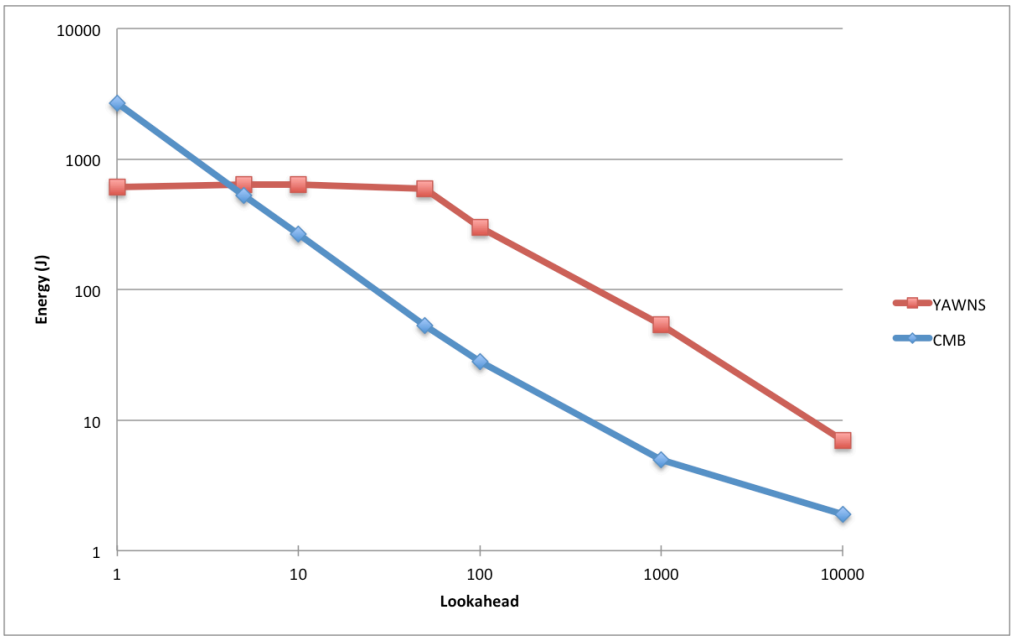

Figure 2-3: Energy consumed for Chandy/Misra/Bryant and YAWNS synchronization.

The CMB algorithm yielded energy consumption that steadily decreased as the lookahead was increased. For very small lookahead values CMB is prone to a phenomenon called lookahead creep where null messages must be sent among the processors to, in effect, enable them to advance by an amount of simulation time equal to

the lookahead. To a first-order approximation, doubling the lookahead value approximately doubles the amount of time advance that can be gained with each "round" of null messages. Assuming the primary cause of energy utilization is the time to send null messages, the data shown in Figure 2-3 is consistent with this observation where it is seen a steady decline in energy consumption results as the lookahead value is increased. Figure 2-4 shows the number of NULL messages sent in these experiments and is consistent with this explanation. It can be seen that the number of NULL messages mirrors the energy consumption data.

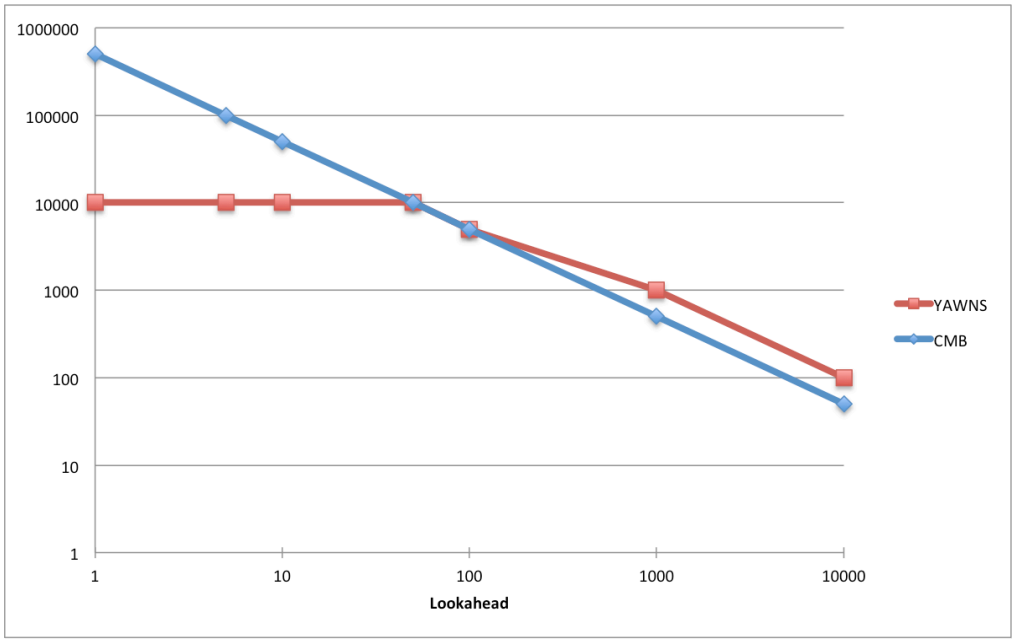

Figure 2-4: Total number of synchronization messages exchanged for the Chandy/Misra/Bryant and YAWNS synchronization algorithms as lookahead is varied.

The YAWNS experiments yielded a decidedly different behavior. Here, the energy consumptions remain at a relatively constant level for small to moderate lookahead values. However, energy consumption then steadily decreases with lookahead

increases at relatively high lookahead values. In contrast to CMB, YAWNS is not prone to the lookahead creep problem in the sense that the algorithm exploits knowledge of the timestamp of the next unprocessed event to advance simulation time. Consider the case where the lookahead is very small, say 1, and the average time between events is 10 units of simulation time. In accordance with time creep, CMB must advance each LP by increments of 1 with each round of null messages to advance LPs to the point where they can process the next (non-null) event. On the other hand, YAWNS will immediately advance the LP to the time of the next event. If the lookahead is small, YAWNS will, again to a first-order estimate, advance LBTS to the timestamp of the next unprocessed simulation event. Therefore, the energy required for synchronization is in proportion to the number of events, independent of the lookahead value, explaining why energy consumption remains flat for small lookahead values. This remains true until the lookahead becomes large. With large lookahead values, many events can be processed in each epoch of YAWNS. Roughly speaking, doubling the lookahead value approximately doubles the number of events that can be processed within each epoch. Thus, for large lookahead values, the energy overhead steadily declines as the lookahead increases. The data in Figure 2-3 is consistent with this explanation. Figure 2-4 shows the number of synchronization messages sent by the YAWNS algorithm. As with CMB, this data mirrors the energy consumption data.

Overall, YAWNS and CMB use roughly comparable amounts of energy in the experiments with large lookahead values. However, these measurements also suggest CMB expends considerably more energy at very low lookahead values due to the lookahead creep problem.

2.5.2 Energy Overhead for Distributed Simulations with Event Communication

The next set of experiments examined the energy consumed when event communications are added. These experiments are like those described in the previous section, but inter-processor communications of events are added in order to assess the energy consumed for synchronization relative to event transmission.

Phold is used for these experiments. In Phold, the probability of remote events (PoRE) is a variable that controls the amount of event communication occurring in a distributed simulation. PoRE is defined as the probability a new remote event will be scheduled when processing an event. Each event processed by the Phold simulation will schedule one new event remotely to an LP on another processor with probability PoRE or locally to an LP on the same processor as the sender with probability 1-PoRE. A distributed Phold simulation with no remote event communication, as in the test case used in the previous experiment, is equivalent to assuming PoRE is zero.

One might initially expect the number of NULL messages exchanged between peers in a peer-to-peer distributed simulation using CMB would decrease with an increase of event communication between the peers because event messages effectively replace NULL messages. To test this hypothesis, we measured an implementation of Phold on the android smartphones. The simulation processes $9x10^5$ events with varying probability of remote events. We use two smartphones as peers for this experiment. Whenever a LP processes an event it updates its current simulation time and then decides whether to schedule the event in the same processor or on a remote processor based on a draw from a uniform random number generator and the PoRE value. While high PoRE

values combined with small granularity would lead to poor speedup in distributed simulations, for real-time applications meeting deadlines is generally more important than speedup, per se. Hence it is difficult to assess what PoRE values will appear in actual applications. As such, we experimented with four different PoRE values spanning the possible range.

Figure 2-5 shows the energy consumed by the distributed simulation and Figure 2-6 indicates the number of NULL messages exchanged between peers for different PoRE values using CMB. The relationship among energy, the number of NULL messages and number of remote event messages is not as straightforward as one might initially think. As the number of remote event messages increases the number of NULL messages exchanged between the processes exhibits a non-linear behavior with respect to PoRE values. At first, additional remote events result in a decrease in the number of NULL messages, as well as a "flattening" of the curves, i.e., less dependence on lookahead. At larger PoRE values we observe the number of NULL messages *increases*. A similar trend is observed for the energy consumed by the simulations as shown in Figure 2-5.

Stated another way, when an LP blocks because it has events to process, but none are safe, it generates NULL messages. These are called Type 1 NULL messages. On the other hand, when an LP blocks because it does not have any events to process, it sends Type 2 NULL messages. For example, consider a peer-to-peer simulation with two peers. One of the peer is blocked due to an empty queue at a simulation timestamp t and the next event to be processed by the other peer has timestamp $t + l$ and the last event

processed by the LP was at t-l', where l and l' are greater than the assumed lookahead value. This deadlock situation would persist in the absence of Type 2 NULL messages.

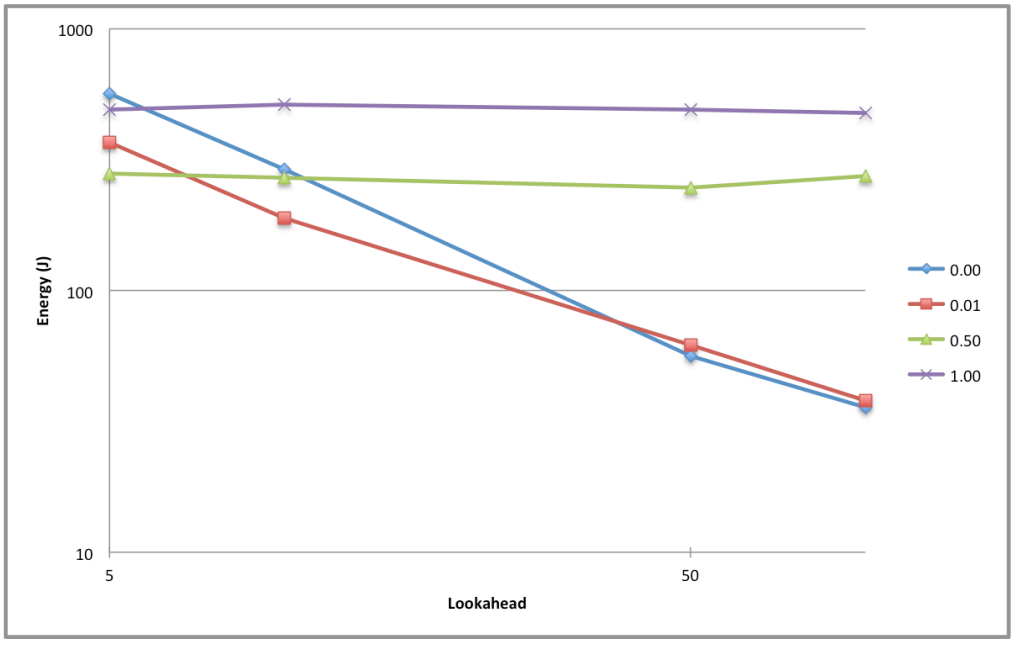

Figure 2-5: Energy consumed by Phold simulation with CMB synchronization algorithm, for varying values of Lookahead and Probability of Remote events. Axes are in Log scale.

Figure 2-6 illustrates the tradeoff between type 1 and type 2 NULL messages. It can be seen that for small values of PoRE, the number of NULL messages decreases with increasing lookahead value, as is expected because for small values of PoRE type 1 NULL messages dominate. The similar behavior of the curves for 0 and 0.01 PoRE values is consistent with this observation.

The flat nature of the curves in Figure 2-6 for higher values of PoRE can be attributed to the dominance of type 2 NULL messages in the simulation, which as mentioned earlier, is caused by not having events to process. Hence, the numbers of

NULL messages for simulations with large PoRE values are not strongly affected by the lookahead value.

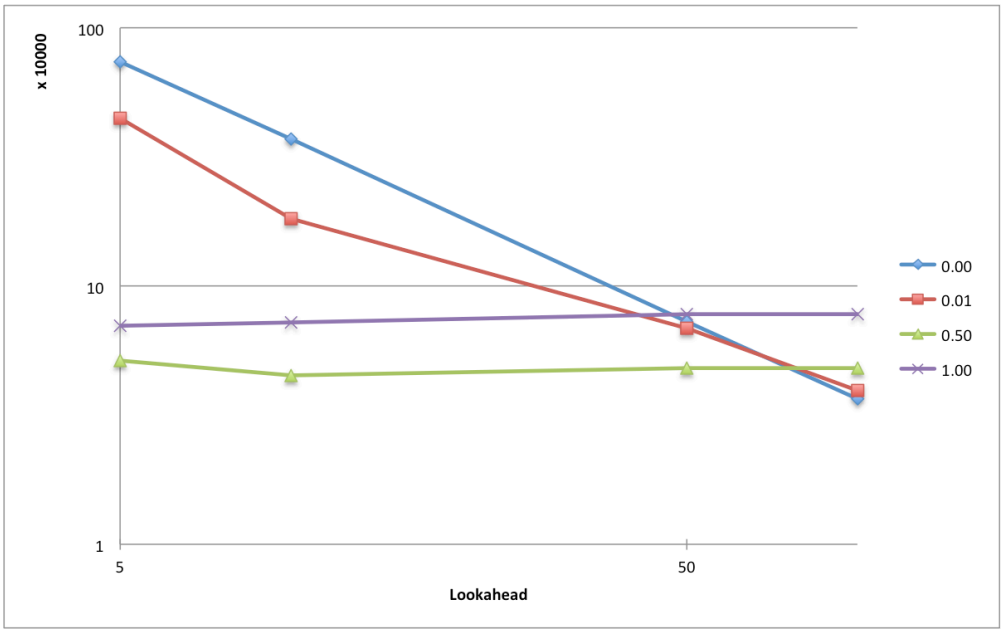

Figure 2-6: Number of NULL messages exchanged among peers in Phold simulation with CMB synchronization algorithm, for varying values of Lookahead and Probability of Remote events. Axes in this figure are in Log scale.

Figure 2-7 shows the number of NULL messages generated as PoRE values are increased for lookahead values of 5 and 100. The sharp decrease in the number of NULL messages for lower lookahead values is expected due to lookahead creep, as discussed earlier. For high PoRE values the number of NULL messages increases slightly because the probability the local event queue is empty increases somewhat with PoRE because there are fewer locally generated messages.

To explain the energy consumption behavior shown by Figures 2-5 and 2-8, we need to consider communications both for events and synchronization. Each curve in Figure 2-5 represents a constant PoRE value, i.e., a fixed amount of remote

communication, so the number of NULL messages largely determines energy consumption. The initial decrease in energy shown in Figure 2-5 as lookahead increases can be attributed to a decrease in lookahead creep. This does not arise at higher lookahead values. We see a similar behavior for low PoRE values at higher lookahead, and little change in energy consumption for high PoRE values as discussed earlier.

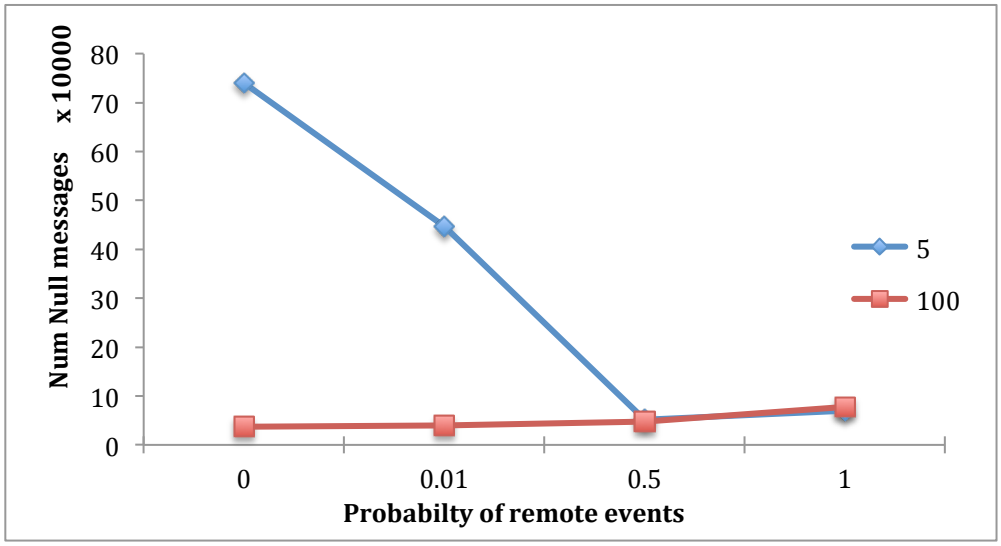

Figure 2-7: NULL messages exchanged by a Phold simulation with CMB synchronization algorithm, for varying values of Lookahead and Probability of remote events.

In Figure 2-8 it can be seen that increasing the PoRE value increases the number of remote messages, which should increase energy consumption. However, the number of NULL messages mitigates this effect. When lookahead creep is an issue, as is the case for lookahead of 5 and few remote events, increasing PoRE reduces the number of NULL messages resulting in a net energy reduction. However, in the other cases where lookahead creep is not an issue, increasing PoRE increases energy consumption because there is more remote event communication.

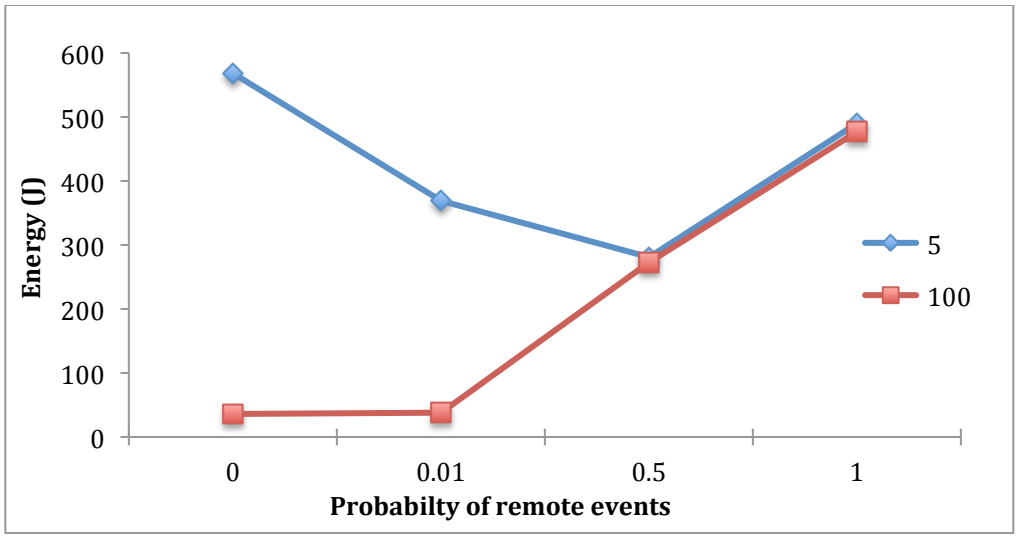

Figure 2-8: Energy consumed by Phold simulation with CMB synchronization algorithm, for varying values of Lookahead and Probability of Remote events.

This analysis of a widely used synchronization algorithm illustrates the unexpected synchronization overhead thus pointing to the importance of a careful choice of the synchronization algorithm even for a simulation with higher event communications.

2.5.3 Queueing Network Simulations

A second benchmark program used in this study is a simulation of a closed queueing network with *J* jobs circulating among the nodes of the network. The queueing network is configured as a three-dimensional toroid topology. Each processor is assigned one two-dimensional plane of the toroid. Once a job receives service it is routed to a randomly selected neighboring node, with each neighbor equally likely to be selected. Each node of the network contains a single server with service time drawn from an exponential distribution plus a constant value *L*. Jobs arriving at each network node are placed into a single queue and are served in first-come-first-serve order. The minimum

service time L is used as a control variable to facilitate experimentation with increased lookahead values. In these experiments the lookahead is enhanced by pre-sampling the random number generator to produce the service time of the next job to be processed by the server; if the pre-sampled value is P, then the time stamp of the next message generated by the LP must be at least $L+P$ units of simulation time into the future (Nicol 1988). Further, the simulation is optimized to exploit the fact that the queue uses a FCFS queueing discipline, resulting in increased lookahead in proportion to the queueing delay. The benchmark program is written in C.

Figure 2-9 shows the energy consumed by the three simulations for queueing networks of size 4 (2x2), 49 (7x7), 484 (22x22), and 1024 (32x32) nodes executing on each processor. The total number of events processed by the simulators was kept constant across all of these runs, i.e., the total amount of simulation computation remained the same across the runs. This figure shows energy consumption data; power consumption data demonstrated similar trends to that shown in the figure.

It can be seen that the energy consumed by the sequential and P2P-CMB simulations remains about constant as network size is increased, but there is a modest reduction of approximately 13% in energy consumed by the CS-YAWNS simulation for the largest network compared to the smallest. We believe this is due to a much smaller number of synchronization messages in CS-YAWNS in the larger sized queueing networks. As discussed earlier, YAWNS operates using a scheme where all events in the current epoch can be safely processed without concern for events later arriving with a smaller timestamp. For these experiments the lookahead, i.e., the minimum timestamp increment, remains the same across all runs. Therefore, there will be more events within a

single epoch that may be processed before the next global synchronization. Since the total number of events in the computation remains the same, this results in fewer global synchronization points. The number of synchronization messages in YAWNS was reduced in approximately the same proportion as the size of the network (a factor of 256) across these experiments.

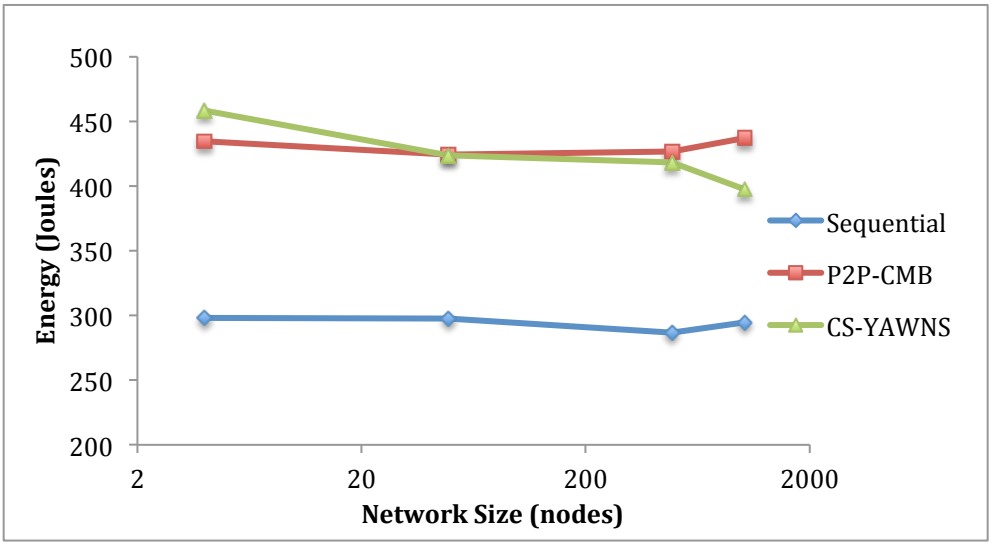

Figure 2-9: Energy consumption for 2x2, 7x7, 22x22, and 32x32 queueing networks. Note network size is plotted on a logarithmic scale.

Varying the size of the network changes the amount of computation performed between communications. Larger networks will have more simulation computations to complete between successive interprocessor message communications, likely accounting for the differences shown in this figure.

The energy overhead resulting from the distributed execution of the simulation program relative the sequential implementation using these two synchronization algorithms and architectures is shown in Figure 2-10 and 2-11. In Figure 2-10, these data are derived by subtracting the energy consumed by the sequential implementation from

the distributed execution with weak scaling, for each network size. For Figure 2-11, the same has been plotted as a percentage of the energy expended in the sequential execution.

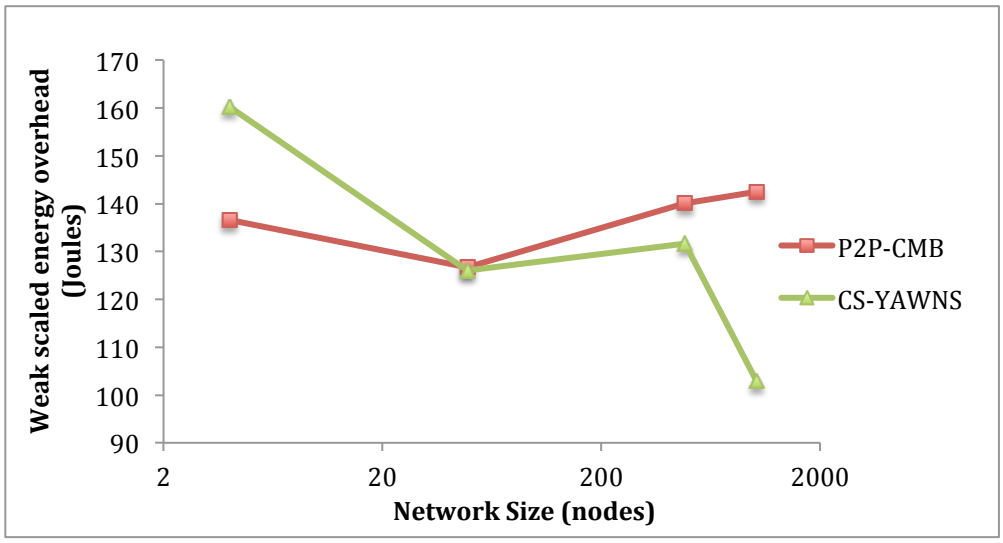

Figure 2-10: Weak scaled energy overhead of CMB and YAWNS for different sized queueing networks.

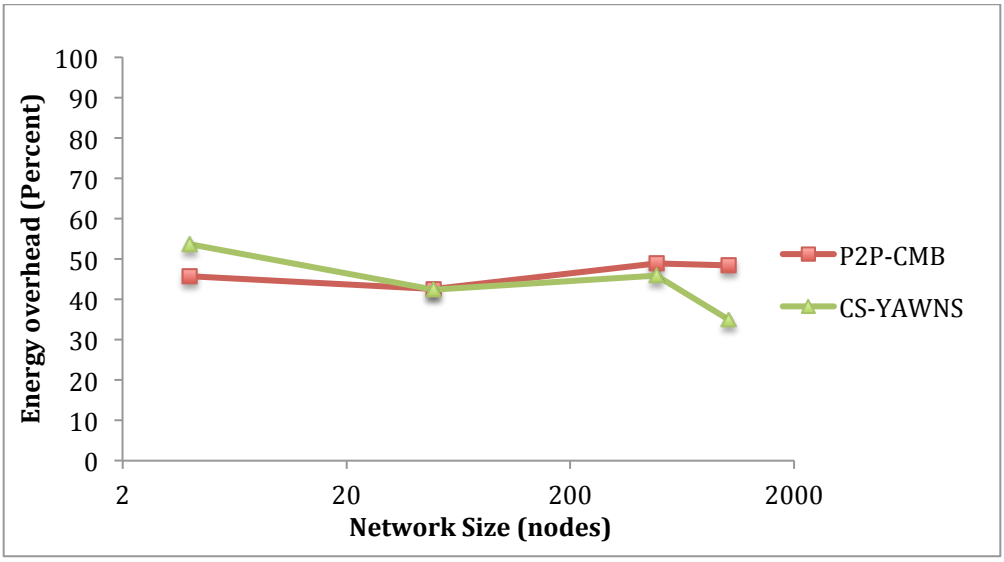

Figure 2-11: Energy overhead of CMB and YAWNS for different sized queueing networks as a percentage of the energy expended by the sequential execution.

As can be seen, the distributed simulations expend from 35% to 54% more energy than the sequential simulation executing the same number of events. While the P2P implementation of CMB remains approximately the same percentage of energy overhead for the different sized networks, there is more variability in the client-server YAWNS implementation.

2.6 Discussion

One can observe a few trends that emerge across these experiments. First, these data highlight that the energy cost resulting from the distributed execution of a simulation program is significant. This observation encourages a further investigation that would allow us to delve deeper into the different functional components of a distributed simulation and their energy consumption. Chapter 3 follows this up with a methodology to look deeper into the energy consumption and provides empirical results.

Second, we observe that different synchronization algorithms and architectures exhibit different behaviors with respect to energy consumption. It is clear that different synchronization algorithms will exhibit different message passing behaviors, so in this sense it is not surprising that they yield different energy consumption characteristics. These data indicate that these differences can be significant and can lead to observable differences in energy consumption. Thus, for applications where energy is a critical factor, e.g., for distributed simulations executing on mobile devices, some care should be taken with respect to the choice of synchronization algorithm in order to maximize battery life.

Third, aspects of the distributed simulation application such as lookahead and event communication that impact the behavior of the synchronization algorithm may have a significant impact on the energy consumption of the distributed simulation. Further investigation is required to examine the impact of aspects such as lookahead on energy efficiency and to gain a deep understanding of this relationship.

When comparing the energy consumption of these two, very different, synchronization algorithms, two observations are apparent. First, to a first order approximation, the overall, average energy overhead observed for these synchronization algorithms is significant, and to some extent, relatively similar. On the other hand, the two algorithms exhibit different energy characteristics as parameters of the simulation such as the number of LPs and lookahead change. One must be cautionary in that these observations are based on a very limited amount of experimental data. Nevertheless, these results suggest that further exploration of the relationship between the synchronization algorithm and energy and power consumption is warranted.

2.7 Conclusions

In this chapter we have argued that power and energy consumption are areas of increasing concern for parallel and distributed simulation systems. Metrics that are consistent and complementary to standard metrics used for time are proposed. Empirical measurements of synchronization algorithms demonstrate that distributed execution incurs a significant overhead in energy consumption and architectural choices significantly impact the amount of energy and power that is consumed.

Power- and energy- consumption of parallel and distributed simulations raise many unanswered questions. Beyond analysis studies, the development and evaluation of techniques to reduce energy- and power- consumption of distributed simulations is an important area of research. These form the focus of the rest of this body of work.

3 PROFILING ENERGY CONSUMPTION OF DISTRBUTED SIMULATIONS

As concluded by the previous chapter, energy and power consumption of computations and communications are areas of increasing concern. Reductions in energy consumption for mobile computing applications will translate to increased battery life and/or reductions in the size and weight of batteries used to power the device. In high performance computing energy has become a major concern because power is a major cost in operating data centers.

There have been a myriad of studies and numerous methodologies have been proposed to measure and predict the energy consumption of specific smartphones applications, but most of these address the problem either within the context of a specific application program (Dong and Zhong 2011, Qualcomm_Technologies 2015) or address low level aspects concerning the hardware, compiler, or operating system (Tiwari, Malik et al. 1996). Energy consumption of distributed simulations have not been widely studied (Fujimoto and Biswas 2015). Very little is known concerning how energy is used in distributed simulations, or simulation applications in general. Clearly such knowledge is required before one can begin to develop approaches to effectively manage and reduce the energy consumed by distributed simulations or to understand tradeoffs among energy consumption, execution time, and model detail and accuracy.

This chapter proposes a model that breaks down energy consumption of distributed simulations into a set of separate functional components. The intent of this

model is to separate various aspects of the distributed simulation in order to quantify the energy consumed by each one, with the eventual goal to inform and guide the development of approaches to effectively manage and reduce energy consumption. We propose experimental methods to measure and separate the energy consumed by the functional components defined by the model and demonstrate their use in conducting an empirical study evaluating the energy consumed by a benchmark distributed simulation program.

The remainder of this chapter is organized as follows. The next section presents related work. This is followed by a description of the model that is proposed for profiling energy consumption in distributed simulations. The section that follows discusses techniques for measuring and separating the energy consumed by various elements of the model. Results of an empirical study exercising these methods are then presented. Data supporting the validity of the experimental approach are presented followed by a discussion of future work.

3.1 Related Work

The bulk of the work concerning energy profiling has focused on predictive modeling of energy and power (Williams, Waterman et al. 2009, Zhang, Tiwana et al. 2010, Keckler, Dally et al. 2011, Czechowski and Vuduc 2013, Mittal 2014) and embedded systems evaluations (Rajovic, Carpenter et al. 2013, Rajovic, Rico et al. 2013, Stanisic, Videau et al. 2013, Grasso, Radojkovic et al. 2014). Model based energy estimation tools are only as good as the training and accuracy of the model. This requirement along with the fact that these models generally depend on the hardware,

usage (Dong and Zhong 2011) and/or platform restricts their applicability to a limited number of devices and/or platforms. A hybrid approach (Dong and Zhong 2011, Bornholt, Mytkowicz et al. 2012) allows the models to be altered based on factors such as hardware and usage. Tools such as Trepn profiler (Qualcomm_Technologies 2015), which rely on on-board power sensors, i.e. a fuel gauge, compute the energy consumed by the complete application but do not separate energy use among different portions of the program. Existing approaches to power or energy profiling do not allow the separation of energy required for all of the functional components of interest in optimizing energy use.

One way to profile the energy consumed by the different functional components is to compute the energy consumption of each machine instruction (Tiwari, Malik et al. 1996, Pathak, Hu et al. 2011). This approach does not extend well to modeling components other than the CPU (Bornholt, Mytkowicz et al. 2012). As will be observed later, the energy consumption of the non-CPU components, e.g., communications can be large compared to the energy consumed by CPU in a distributed simulation. In addition, the instruction-based approach suffers from the drawback of requiring hardware specific values that may not be readily available.

There have been only a small number of studies concerning the energy and power consumption of parallel and distributed simulations. In (Shi, Perumalla et al. 2003), the authors studied power consumption for disseminating state information in distributed virtual environments. This study highlighted dead-reckoning algorithms and tradeoffs between state consistency and power consumption. Studies described in (Feng, Ge et al. 2005, Ge, Feng et al. 2010, Dongarra, Ltaief et al. 2012, Esmaeilzadeh, Cao et al. 2012)

focus on characterizing power consumption for scientific computing applications. Authors of (Child and Wilsey 2012) presents a preliminary study of energy consumption of Time-warp simulation on a shared memory multi-core processor with Dynamic Voltage and Frequency Scaling. The work described in (Neal, Kanitkar et al. 2014) presents a study of power consumption of data distributed management (DDM) services defined in the High Level Architecture. One recent work examined the power and energy consumption of synchronization algorithms for parallel and distributed simulations (Fujimoto and Biswas 2015). To the best of our knowledge there has been no study that breaks down the energy consumption of a distributed simulation into its functional components, the primary focus of this work.

The approach proposed here does not assume or require any hardware or platform specific details and relies only on the availability of a coarse power measurement tool for the platform of interest that reports overall energy consumption for an entire application program. Numerous hardware and software platforms are currently available that have this energy measurement capability. Studies report that software tools such as the Trepn Profiler (Qualcomm_Technologies 2015) used here indicate they can be as accurate as methods based on external hardware measurement devices such as the Monsoon power meter, but at a much lower cost (Schwartz 2015).

3.2 A Model for Energy Consumption in Distributed Simulations

As discussed earlier, the purpose of the proposed energy model is to separate energy consumption for different aspects of distributed simulations programs. This model can be used to create profiles indicating the amount of energy consumed by different

portions of the distributed simulator to pinpoint where improvements may have the greatest impact in reducing energy consumption. As is standard practice, we assume the distributed simulation application consists of a set of logical processes (LPs) that communicate by exchanging timestamped event messages.

We first differentiate between the *simulation engine* and the *simulation application*. We informally define the simulation engine as those portions of the distributed simulation that are fixed across all simulation applications. The simulation engine includes functions such as managing the execution of LPs, event lists, and synchronization. It also includes interprocessor message communications both for synchronization and exchanging application event messages. The simulation application includes all other software, and includes all software associated with the simulation model. Library functions such as random number generators are assumed to be part of the simulation application. While the distinction between the simulation engine and application is somewhat subjective for some simulators, distributed simulation systems typically define an applications program interface (API) that provides a clear separation between these two components. For example, in federated distributed simulation architectures, all software associated with the runtime infrastructure is considered part of the simulation engine, as well as much of the software within the federate such as that associated with event list management. We denote the energy consumed by the simulation engine by E_{SE} and $that$ consumed by the application by E_{app}.

We can further divide software within the simulation engine into two major functional components: that implementing communication functions and that associated with simulation engine computation. Let E_{comm} denotes the energy consumed by

communication functions and E_{SEcomp} denotes energy associated with computations performed within simulation engine, e.g., event list management and LP scheduling. We assume the simulation engine performs all interprocessor communication. These communications include that required for sending and receiving event messages and other overhead communications, e.g., the communications required for synchronization. E_{comm} may be further broken down into the energy consumed for sending and receiving messages, denoted E_{send} and E_{rcv}, respectively, however, here we will be content to focus only on measuring E_{comm}. N_{event} indicates the total number of event messages received/sent by a node. Similarly, the number of messages transmitted for overhead functions, especially synchronization is defined as N_{sync}.

The total energy consumed by the distributed simulation (sequential simulations are viewed as a special case where E_{comm} and all its components are zero) is denoted E_{sim}. It is easy to see the relationships among the components making up our energy profile below.

$$E_{sim} = E_{SE} + E_{app} \quad (1)$$

$$E_{SE} = E_{comm} + E_{SEcomp} \quad (2)$$

$$E_{comm} = E_{rcv} + E_{send} \quad (3)$$

$$N_{comm} = N_{event} + N_{sync} \quad (4)$$

$$N_{event} = N_{eRcv} + N_{eSend} \quad (5)$$

$$N_{sync} = N_{sRcv} + N_{sSend} \quad (6)$$

3.3 Constructing Energy Profiles

An energy profile specifies the amount of energy consumed by each of the components defined in the model discussed in the previous section. The value of energy profiles is clear; just as time profiles are commonly used in optimizing the performance of software by revealing bottlenecks in a code, energy profiles may be used to guide optimization of energy use by revealing energy hogs. However, time profiles are easily constructed using a high-precision real-time clock or using sampling techniques. Constructing energy profiles is not so straightforward. One reason is because the hardware platform may not provide mechanisms to perform high precision, low overhead energy consumption measurements. Energy consumption can be computed from instantaneous power, which in turn can be computed as the product of instantaneous current used by the circuit and the voltage. However, reading the instantaneous current is often a system call that requires a significant amount of overhead when attempting to obtain high precision measurements. More seriously, even if high precision, low overhead current values can be obtained, the measured current includes that used by all hardware components including the CPU, memory system, and networking circuits that are all operating concurrently. There is no simple way to separate these. Models may be used to estimate the energy consumed by different hardware components, however this raises questions concerning the validity and applicability of the models, as was discussed earlier.

This section focuses on defining techniques to create energy profiles of distributed simulations with coarse energy measurement mechanisms. In the following section, we describe a methodology for measuring three major components of a

distributed simulation, namely communication (*i.e.* E_{comm}), simulation engine computation (E_{SEcomp}) and application (E_{app}).

3.3.1 Measuring Energy for Communication

The main challenge in measuring the energy used for communication is to separate this energy from that used for computations not related to the communications. A technique is proposed that involves creating and measuring the energy consumed by a *pseudo distributed simulation*, or simply a pseudo simulation, that excludes all communication as discussed below. The total energy consumed by the distributed simulation less that used by the pseudo simulation yields the amount of energy consumed for communications.

The *pseudo distributed simulation* must repeat the same computations performed by the distributed simulation execution but with all communications used by the distributed simulation removed. This is accomplished by first executing the complete distributed simulation in order to generate a log of all messages sent and received by the computation. The distributed simulation that includes all portions/aspects of the execution is called the *original simulation*. Coarse energy measurement tools can be used to measure the energy consumed by the original simulation. Then the distributed simulation on each processor is repeated without any interprocessor communication. No operation is performed when a message send is executed. Each time a message is received in the pseudo simulation the appropriate message is retrieved from the log rather than performing actual communications. The energy used by the pseudo simulation reflects the energy used by the computation part alone without energy expended for

communication, and coarse energy measurement tools can be used to measure the overall energy consumption. Retrieving messages from the log consumes some energy, however, as will be seen later, the amount is small and can be ignored.

Stated more succinctly, let the energy of the pseudo distributed simulation run be denoted by $\mathrm{E}_{\mathrm{pseudo}}$ and that of original simulation run be denoted by $\mathrm{E}_{\mathrm{orig}}$, then we can write the following:

$$\mathrm{E}_{sim} = E_{comm} + E_{pseudo} \quad (7)$$

$$\mathrm{E}_{sim} = \mathrm{E}_{\mathrm{orig}} \quad (8)$$

From Equation 7 and 8, we have

$$\mathrm{E}_{comm} = \mathrm{E}_{\mathrm{orig}} - \mathrm{E}_{\mathrm{pseudo}} \quad (9)$$

The message communication log is obtained from the original simulation run. It should be noted that the message communication log must be generated in a separate execution of the original simulation, as otherwise the energy consumed by the I/O operations required for the generation of the log would figure in $\mathrm{E}_{\mathrm{org}}$. The log must contain sufficient information to ensure each message is delivered at the correct point during the re-execution, i.e., in the execution of the pseudo simulation. It is sufficient to log (1) the contents of the message, (2) the simulation time stamp assigned to the event, and (3) a temporal value indicating the point during the execution when the message is delivered. In the experiments described later the simulation executive contains a loop that is concerned with processing events; the iteration number for this loop is used to specify

this third value. By introducing necessary wait times, which might arise due to communication delays and non-uniform processing power of the nodes, this ensures that the run-time behavior of the pseudo simulation is similar to that of the original simulation. Note that the log includes both, event messages as well as messages used for synchronization or other purposes. The message log is stored in secondary storage for use by the pseudo simulation.

When the pseudo simulation is executed the log of received messages is read and stored in memory during the initialization step of the pseudo simulation. This initialization step must be excluded from the measurements of energy consumption by the pseudo simulation. Loading the message log into memory during the initialization phase assumes there is sufficient memory to hold the entire log. If this is not the case one could load the message log into memory incrementally as needed during the execution of the pseudo simulation, however, one must be sure to exclude the energy consumed in reading the log into memory from the measurements of energy consumed by the pseudo simulation.

After the message log has been loaded, it is used to replace incoming messages received from other processors during the execution of the pseudo simulation run. This is achieved by incorporating (i.e. perform all the computations as if a real message was received) the i^{th} message from the data in the *pseudo simulation run*, when the current simulation time of the pseudo simulation run is either greater than or equal to current_simulation_time of the original simulation associated with the i^{th} message. This assures that the computations performed by the pseudo simulation are the same as that of the original execution.

This approach assumes the energy consumed to retrieve messages from the message log is negligible relative to other operations performed by the simulation. This is a reasonable assumption because the operations to retrieve messages from the log are simple and consume little time. This assumption was validated through experimentation, as discussed later.

Message send operations typically do not affect the computations performed by the application or simulation engine so they can be ignored in the pseudo simulation. In situations where the computation is affected by the message send, e.g., use of values returned by the send function, one would need to also log the results of these operations as well in order to ensure the pseudo simulation correctly reproduces the behavior of the program. Computations leading up to the sending of the message, e.g., buffer allocation and constructing the message itself are included in the computation portion of the energy consumption profile.

3.3.2 Measuring Energy Consumed by Simulation Engine Computations

Once the energy consumed for computations has been separated from that utilized for communication, the next step in the profiling process is to separate the energy consumed by the simulation application from that utilized by the distributed simulation engine. The main challenge is devising a way to separate energy consumed by application computations from that expended by engine computations.

One approach to separate application and engine computations is to start with the pseudo distributed simulation as described earlier and then delete all application code, leaving only the simulation engine code. Note the communications is also absent since

we started with the pseudo simulation. The resulting code can be executed, and its energy consumption directly measured. With this approach interactions between the simulation engine and the applications, e.g., function calls made by the application to the simulation engine must be logged, as well as the parameters used in those calls. The log is referenced when the engine is executed in isolation in order to ensure that the behavior of the simulation engine is correctly reproduced when it is executed without the simulation application.

A drawback of this approach is the energy consumed by the simulation engine executing in isolation may be different from that when it is executing with the application. While the simulation engine will execute the same machine instructions with or without the application being present, the timing and detailed operation of the CPU and memory system may not be the same. For example, if the application consumes a large amount of memory, the application may allocate cache memory and cause cache misses to occur within the simulation engine, an effect that is lost when the engine is executed in isolation without the application. One might argue that a similar error could arise when executing the pseudo simulation relative to the original distributed simulation because the communications part is missing, however we believe this impact will be less significant than an application consuming large amounts of memory.

In order to capture effects such as this, an alternative approach to measuring energy consumed by the computation portion of the simulation engine was developed. We start with the pseudo simulation that includes both the application and engine code. We add an additional copy of the engine code and associated data structures. Now, when the pseudo simulation is executed, we execute each engine operation *twice*, once as part

of the normal execution of the pseudo simulation, and once in the replicated engine. We refer to this code as the *replicated engine code*. The replicated simulation engine code computes, or in other words decides its path of execution, based on the original state variables. This removes the requirement of replicating any computation that lies outside the simulation engine. It should be noted here that although the replicated simulation engine compute based on the original state variables, it only updates the replicated/dummy variables. The energy consumed by the replicated engine code is denoted as $\mathrm{E_{SEcompX1}}$. The subscript *SEcompX*1 indicates one additional execution of the simulation engine computations is included in the measurement. The amount of energy consumed by simulation engine computations is simply $\mathrm{E_{SEcompx1}} - E_{pseudo}$, where $\mathrm{E_{pseudo}}$ is the energy consumed by the pseudo simulation including the application and one execution of the engine computations.

The introduction of a replicated copy of the simulation engine may have some impact on the execution of the original pseudo simulation because it too will utilize the memory system, introducing some error. We believe this impact will be small if the engine has a relatively large memory footprint, which is generally the case for real-world simulations. This would ensure that the misses remain relatively consistent even with replications. Empirical evidence supporting this claim is presented later.

It may be noted that this approach using a replicated engine avoids the need to generate a log of calls to the simulation engine. This simplifies the development of the instrumentation code.

In a well-designed simulation engine, the amount of computation used by the engine may be modest. To enable accurate energy measurements of the pseudo engine using coarse energy measurement tools one can simply execute *k* replications of the engine. Let $\mathrm{E}_{\mathrm{SEcompXk}}$ be the energy consumed by the pseudo simulation augmented with *k* replications of the engine. The energy used for simulation engine computations is easily computed as:

$$\mathrm{E}_{SEcomp} = \frac{\mathrm{E}_{SEcompXk} - E_{pseudo}}{k} \quad (10)$$

3.3.3 Measuring Energy Consumed by the Application

The methodology presented in the previous section can be used to measure the portion of the energy consumption due to executing application computations. For example, one could take the original distributed simulation and create *k* additional replications of the application code. Let $\mathrm{E}_{\mathrm{appXk}}$ denote the energy consumed by this code, including message communications. If one subtracts the energy consumed by the original distributed simulation and divides the result by *k* the result is the power consumed by a single instance of the application code alone:

$$\mathrm{E}_{app} = \frac{\mathrm{E}_{appXk} - E_{sim}}{k} \quad (11)$$

Alternatively, once could use the pseudo simulation as the baseline. In this case the energy consumed by the application code is:

$$\mathrm{E}_{app} = \frac{\mathrm{E}_{appXk} - E_{pseudo}}{k} \quad (12)$$

It may be noted that E_{appX1} (*i.e.* E_{appXk} for k = 1) is not equal to E_{app}, because the former includes the energy of the baseline simulation (distributed simulation or *pseudo distributed simulation* depending on the implementation) in addition to one extra application part. Similar comments apply to $E_{SEcompX1}$ and E_{SEcomp}.

3.4 Empirical Evaluation

A series of experiments were conducted to exercise the proposed methodology and validate the measurements. In this section we describe the setup used to empirically evaluate the energy model and methodology.

3.4.1 Distributed Simulation System

In this study, we use an implementation of a parallel discrete event simulation (PDES) program using the CMB (Chandy/Misra/Bryant) synchronization algorithm (Bryant 1977, Chandy and Misra 1979). LPs communicate directly with other LPs by sending messages to the appropriate nodes.

3.4.2 Experimental Configuration and Benchmark Application

The experimental configuration is intended to mimic an embedded simulation application where the distributed simulation executes within a set of mobile processors. Cellular phones are used for these experiments. The processor used in these phones is typical of what one might anticipate in future deployments of embedded distributed simulations. Specifically, a LG Nexus 5 cellular phone with a quad-core Qualcomm MSM8974 Snapdragon 800 processor, 2 GB memory, and 16 GB storage was used as the mobile computing platform. The phone runs the Android version 5.1 (Android Lollipop)

operating system and was used in the experiments. Hardware-based techniques to reduce power consumption such as voltage or frequency scaling were not used in these experiments.

All inter-processor communications utilize wireless links. The device's 802.11n Wi-Fi network interface was used for communications between processors. A private wireless network was established among the devices to avoid interference resulting from Internet traffic. The cellular network capability of the phone is not used in these experiments.

Queuing networks provide an intuitive, efficient way to study systems that have waiting as a fundamental component. Hence it forms a building block of a very wide range of real-world discrete event simulation applications, e.g. network simulators such as NS3 (NS-3_project 2011) and DDDAS applications such as traffic simulators.

The benchmark program used in this study is a simulation of a closed queuing network with J jobs circulating among the nodes of the network. The queuing network is configured as a three-dimensional toroid topology. Each processor is assigned one two-dimensional plane of the toroid. Once a job receives service, which includes creating a temporary priority queue, pushing J jobs into it, and then popping them out and destroying the queue, it is routed to a randomly selected neighboring node. Each node of the network contains a single server with service time drawn from an exponential distribution. Jobs arriving at each network node are placed into a single queue and are served in first-come-first-serve order. In these experiments the lookahead is enhanced by pre-sampling the random number generator to produce the service time of the next job to

be processed by the server; if the pre-sampled value is P, then the time stamp of the next message generated by the LP must be at least P units of simulation time into the future (Nicol 1988). The benchmark program is developed as a native android application.

3.4.3 Power Measurement Methodology

The energy and power consumption data are derived from direct measurement of the Android device. More specifically, Qualcomm's android app, Trepn (Qualcomm_Technologies 2015) was used for profiling. All the benchmark experimental results presented here are with *Deltas* enabled. When profiling with *Deltas* enabled the app profiles (collects power data) for the entire system for a baselining interval and then subtracts the average value of power, so obtained, from all subsequent raw values. All the experiments were conducted with a maximum possible baselining period of 30 second with wake lock for the entire period of profiling. The power profiles of the application were saved as comma separated value (csv) files, which were then processed offline to compute the energy consumed.

Table 3-1: Measured values for peer of interest in original simulation.

Events processed	99803
Sync messages received (N_{sRcv})	498
Sync messages sent (N_{sSend})	12515
Event messages received (N_{eRcv})	30341
Event messages sent (N_{eSend})	20104

3.4.4 Overall Energy Consumption

The energy consumed by the distributed queuing network simulation is computed by profiling a peer. The peer, denoted as the peer of interest, is simulated with 32*32 queues, each initialized with 10 jobs, for a simulation time of 200000 time units. Table 3-1 presents values measured from the original experimental run.

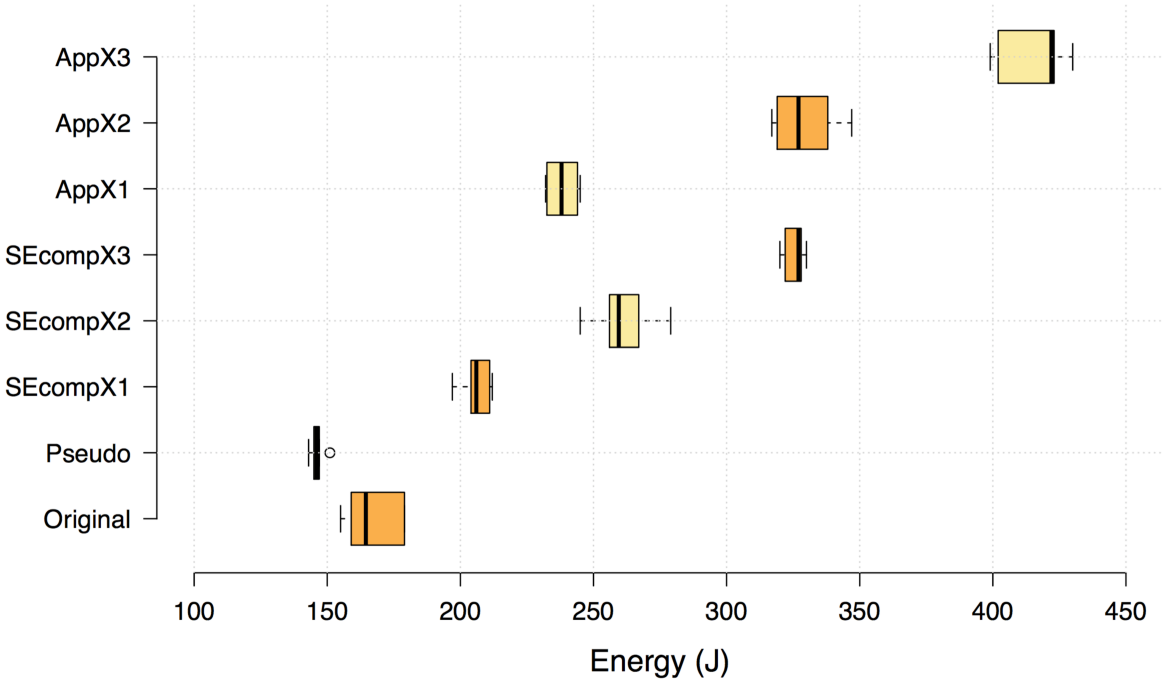

Figure 3-1: Variance in the experimental values of the observable metrics.

3.5 Results

In this section, we start by presenting the energy consumed by each of the components of the distributed simulation (as described in Section 3.4.4 having the features listed in Table 3-1) as determined by the proposed methodologies discussed in Section 3.3 and their corresponding implementations presented in Section 3.4.

Multiple runs were conducted for each metric. Figure 3-1, generated using *BoxPlotR* (Tyer_labs and Rappsilber_labs 2014), presents the ranges of measured data that were observed for each metric. Table 3-2 presents the average values of the runs for each observable metric. We denote the values of components, which can be observed directly as a result of an experiment as *Observable metric* and those which have to derived using *Observable metrics* as the *Derived metrics*.

Table 3-2: Values of the observable metrics.

Metric	Avg. Value (J)
$\boldsymbol{E_{sim}}$	166.83
$\boldsymbol{E_{pseudo}}$	146.33
$\boldsymbol{E_{SEcompX1}}$	206
$\boldsymbol{E_{SEcompX2}}$	261
$\boldsymbol{E_{SEcompX3}}$	325.4
$\boldsymbol{E_{appX1}}$	238.25
$\boldsymbol{E_{appX2}}$	325.6
$\boldsymbol{E_{appX3}}$	415.2

There are three derived metrics of interest: the energy consumed by communication (E_{comm}), the energy consumed by the simulation engine (E_{SEcomp}), and the energy consumed by the simulation application (E_{app}). As indicated in equation 9, E_{comm} can be derived from the observable metrics E_{sim} and E_{pseudo} and is 20.5 J.

The values of E_{Ecomp} and E_{app} can be computed using equation 10 and equation 12, respectively. Figure 3-2 shows the values of the derived metrics for different

numbers of replications (k). It can be seen that varying k for these small values results in little variation, consistent with expectations.

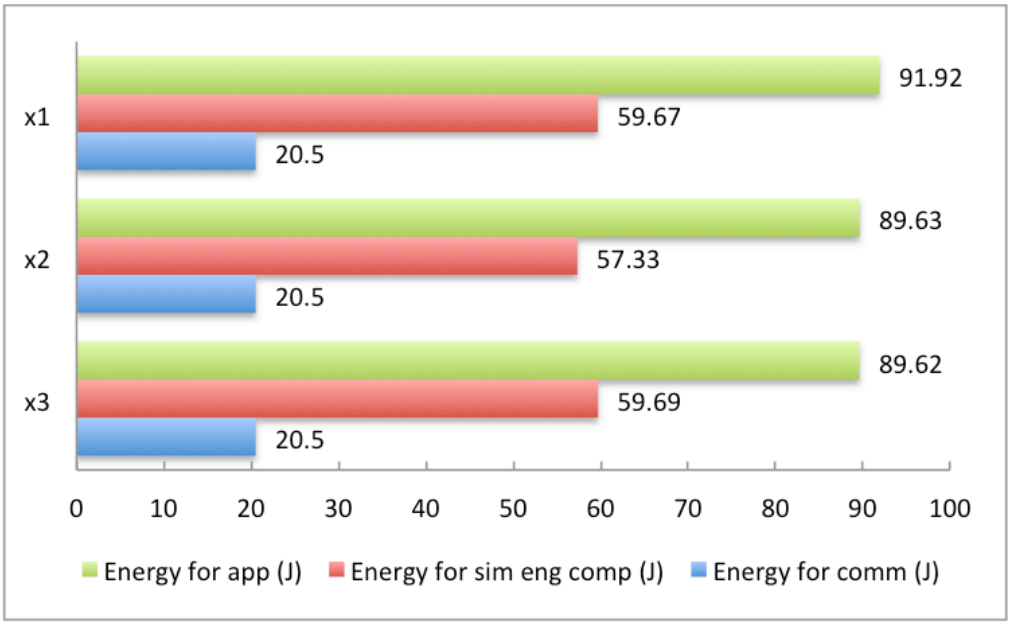

Figure 3-2: Aggregated results comparing energy profiles shows consistency in results of the proposed methodology for different values of *k*.

3.6 Empirical Validation

In this section we present two sets of validation experiments. Micro-validation experiments are used to validate the results obtained for individual parts of the proposed methodology. Macro-validation is used to validate the aggregated energy measurements made using the proposed approach.

3.6.1 Micro-Validations

3.6.1.1 Validating E_{comm}

For the purpose of validating the energy required for communication, we use micro-benchmarking. Another approach to determine E_{comm} is to view it as a sum of

energy required for sending and receiving individual messages. A program was constructed that only performs message sends and receives. This approach ignores interference between the rest of the computation and the message sending and receiving code and uses a simplified model for the messages themselves, as discussed later. This limits the generality of this approach as a means of computing E_{comm}, but is adequate for these experiments.

To determine the energy required for dispatch of an individual message (Here we assume the energy required for a synchronization message reception and dispatch is the same as that of an event message, but this methodology can be easily extended for asymmetric cases as well.), we compute the energy required, say E_{rcvXm}, for sending m messages and then divide it by m. That is,

$$E_{send} = \frac{E_{sendXm}}{m} \tag{13}$$

Similarly, the energy requirement of receiving a message can be computed as follows.

$$E_{rcv} = \frac{E_{rcvXm}}{m} \tag{14}$$

Assuming E_{rcv} denotes the energy required for receiving a message and E_{send} energy required for sending a message, we have

$$E_{comm} = E_{rcv} \text{ x } (N_{eRcv} + N_{sRCV}) + E_{send} \text{ x } (N_{eSend} + N_{sSend}) \tag{15}$$

To implement the micro-benchmark the simulation code is modified to perform two functions. First, the simulation sends some fixed number of randomly generated messages using structures and sizes that are representative of the messages sent in the

distributed simulation to other processors without actually simulating anything. Second, the micro-benchmark simply receives messages and does not process them.

To determine the energy consumed in sending messages the micro-benchmark simply sends the messages but does not receive any. Energy required for sending a message can be derived by using equation 13. Similarly, to determine the energy used to receive messages the processor only receives the *m* messages. The energy required to receive the messages can be determined using equation 14.

Using a value of *m* equal to 250,000, we find the average value of $E_{rcvX250k}$, over multiple runs, to be 28 J and that of $E_{sendX250k}$ to be 108.5 J. From equation 14 and equation 13, it follows that E_{rcv} is 1.12×10^{-4} J and E_{send} is 4.34×10^{-4} J.

Substituting these values into equation 15 along with the values shown in Table 3-1, we obtain E_{comm} is 17.61 J. This compares with a value of 20.5 J for E_{comm} obtained using the pseudo simulation. A somewhat higher value using the pseudo simulation is expected because it the communications executes in conjunction with the simulation engine and application codes.

3.6.1.2 Validating E_{SEcomp} & E_{app}

Energy measurements using the pseudo simulations for different values of *k* were presented in Figure 3-2. It can be seen that the values of E_{SEcomp} varies by at most 4% for different values of *k*. This adds confidence that the methodology produces consistent measurements of application and simulation engine computations. Similarly, comparing the values of E_{app} for different values of k, we can see that they vary by at most by 2.5%.

3.6.2 Macro-Validation

To test the overall model for consistency we add the energy consumed by individual components, as determined by the proposed methods, and compare the result with the total amount of energy consumed by the distributed simulation that was measured, i.e., E_{sim}. Three major components of E_{sim} are energy for communication (E_{comm}), energy for computations of simulation engine (E_{SEcomp}), and the energy for the application (E_{app}). From equations 1 and 2, E_{sim} can be expressed as:

$$E_{sim} = E_{comm} + E_{SEcomp} + E_{app} \quad (16)$$

Table 3-3 compares the observed value of E_{sim}, i.e., 166.83 J, with the value of E_{sim} computed by equation 16. It can be seen that the error is small.

Table 3-3: Validating energy model by comparing computed values of E_{sim} with its observed value.

k	Sum	Error
1	172.08 J	3.15%
2	167.47 J	0.38%
3	169.81 J	1.78%

Figure 3-3 graphically shows the energy profiles of the distributed simulation code for three values of *k*. These graphs illustrate the amount of energy consumed for communications, simulation engine computations, and application computations. This profile illustrates that for this benchmark all three components consume significant amounts of energy, however, that used for computation dominates that used for

communication. Of course, these profiles will vary from one application to another depending on the intensity of communication required by the application.

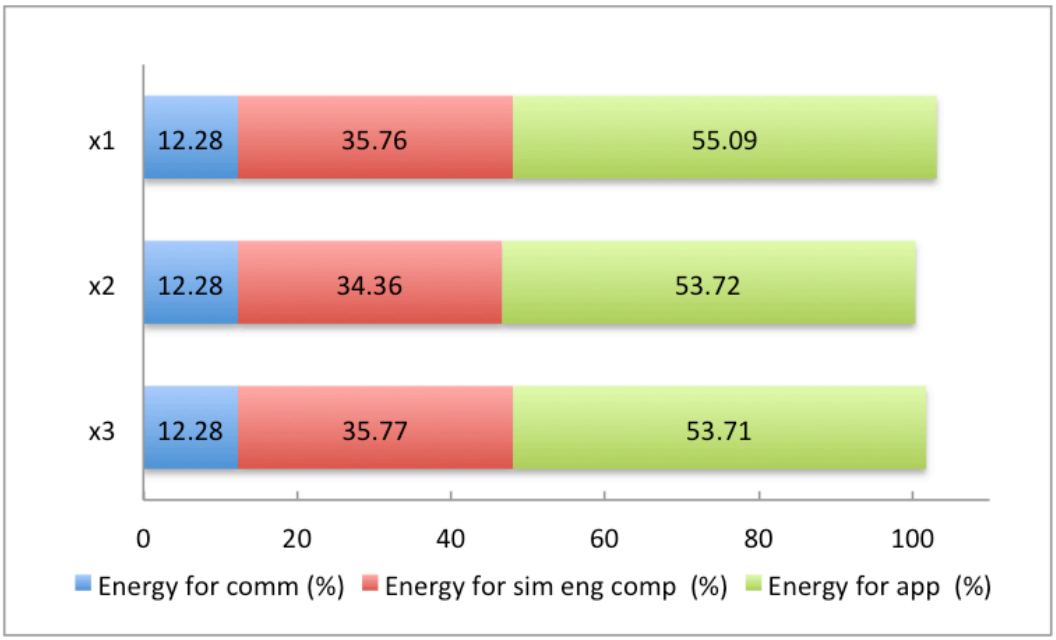

Figure 3-3: Energy measurements with respect to the observed value of E_{sim}, shows that energy computed for individual parts of the simulations adds up very close to total energy consumed by it.

3.7 Conclusions

In this chapter we propose techniques to profile the energy consumed by major functional components of distributed simulation programs. Such profiles are needed to analyze and guide the development of energy optimization techniques. The methodology is followed by an empirical study, demonstrating use of the proposed methodologies and empirically validating the proposed energy model. Although this study focused on conservative synchronization algorithms, we believe the proposed methodology could be extended to apply to distributed simulation systems using optimistic synchronization algorithms. This is one area meriting further investigation.

One important observation from these measurements is that the energy overhead introduced by the distributed execution of the queueing network simulations examined here is significant. Specifically, the distributed simulation engine and communication functions consumed a significant portion of the total energy required to execute the simulations. These preliminary measurements suggest that techniques to optimize the energy costs associated with the distributed execution of simulations may yield significant benefits.

4 ENERGY EFFICIENT DISTRIBUTED MIDDLEWARE

The past decade has seen an enormous increase in the development and adoption of sensors in all walks of life. This has been in conjunction with the widespread deployment and adoption of wireless networks and the Internet leading to an explosion of interest in systems composed of sensors or sensory data of various types and modalities. Such multi-modal systems can perceive the physical world better than ever before. Paradigms such as the Internet of Things (IoT), fog computing, cyber-physical systems, and dynamic data driven application systems are examples of some areas born out of these developments. These emerging systems differ substantially from the systems of the past, both in terms of the possibilities they offer as well as their requirements, especially regarding scale and energy efficient operation.

For example, connected vehicle systems include travelers' mobile devices, in-vehicle onboard computing systems, roadside infrastructure embedded in traffic signal controller cabinets and signs, and centralized local and regional traffic management centers coupled through vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communications. Such systems are transforming urban transportation. A wide variety of applications exist or are under development that will improve safety (e.g., hazard warning or collision avoidance), reduce congestion and traveler delays (e.g., route recommendation, adaptive signal control, event management), and enhance traveler experiences (e.g., entertainment, traveler information systems). Disruptive technologies such as automated vehicles and commercial drones as well as new paradigms such as ride-share systems create new opportunities and challenges for mobile sensor networks in

connected vehicle deployments. An amalgamation of these can improve the efficiency or robustness of the system and lead to improved safety and quality of life for citizens.

These systems may utilize predictive simulations and data analytics software to control and manage the system. The simulations will use historical data stored in back-end databases and rely on dynamic real-time data from in-vehicle and roadside sensor networks to derive information about the current and future states of the transportation network to inform and evaluate decisions. Such an ensemble of technologies and devices enable better-informed decisions and allow for more autonomous systems with little or in some cases no human supervision to manage the transportation network.

Dynamic Data Driven Application Systems (DDDAS) (Darema 2004), as presented in chapter 1, are systems that can dynamically change their behavior based on the state of the environment in which they operate by incorporating real-time data into computations used for decision making. The evolving and sometimes unpredictable nature of systems such as power grids and vehicular traffic make them well suited to exploit the DDDAS paradigm. Example DDDAS applications utilizing computations embedded within the physical system were described in (Blasch, Ravela et al. 2018). Tools are needed to support both the development and deployment of systems such as these. The goal of the G-RTI project was to develop a common middleware platform to support research, development and deployment. In the early phases of design, simulation models are used to represent system components. Development of hardware devices and software to be deployed within the system itself requires an emulation capability where real-world devices are intermixed with simulation models and execute in real time. For example, new apps executing on mobile devices must be tested in the laboratory under

realistic operating conditions that include aspects such as packet loss in wireless networks. Replacing the simulated elements with operational systems provides a natural pathway to transition the system developed in the laboratory to a system that can be deployed in the field. The system development can be greatly accelerated if a single software environment and tool suite can be used to support all aspects of simulation, emulation, and deployment. Such an interfacing of real and simulated worlds arises in agile system development. As an application is being developed, simulations can be used to consider what if scenarios to inform designers regarding revisions of the system (Fujimoto, Barjis et al. 2018).

These systems are composed of multiple sub-systems that must be integrated to form a single seamless system. It is not uncommon to find such systems are composed of components ranging from small individual sensors and hand held mobile devices to large stationary back-end infrastructure. Middleware is required to support the wide variety of devices and technologies that are needed. Such middleware should provide a means to quickly and easily interface the components of the system. Such an array of devices utilizing static and real time data makes development and study of such systems difficult. Any study involving such systems would involve multiple simulators, testbeds and data sources. For example, a connected vehicle application might need a mobility (traffic) simulator, a roadside sensor network operating in the laboratory, and a wireless communication network simulator. Hence there is a need to connect and interface these individual components to enable them to interoperate.

Further, many devices, e.g., mobile devices, operate using batteries as their source of power. As such, energy consumption is a major concern. Middleware software and

algorithms should be designed to minimize the energy consumption of these devices as well as support techniques for reducing energy use in computation, communications and synchronization. The Green Runtime Infrastructure (G-RTI) middleware was developed to address these challenges. It is intended to support all phases of system development including distributed simulation, emulation, and deployment, while providing a common, simple interface to support a wide variety of devices and services. An implementation of G-RTI is available as open-source software. We envision the use of G-RTI would allow for a growth in number of cross boundary connected application and studies by making it simple to develop and test systems that can derive intelligence, actuation and data from a wide variety of sources and simulations.

G-RTI is differentiated from other middleware in several respects. First, G-RTI is designed to support a wide array of heterogeneous systems including IOT and fog or edge computing systems. This introduces several considerations. It must provide flexibility to system developers to dynamically choose the amount of resource an end point possesses, allowing the system to be heterogeneous and aware. The system must support both pub/sub and pull/push based communications. In addition, the server should be able to support on-server applications. As mentioned earlier G-RTI aims to support all phases of system development, including simulation, emulation, and deployment. As such, G-RTI represents a different category of the middleware compared to those listed in (Ngu 2017). G-RTI maintains the advantages of the classes listed in this survey while remaining application independent. Finally, G-RTI allows dynamic addition and removal of clients (simulations or data sources) and objects. This is of importance for interfacing and

maintaining simulation models supporting/simulating dynamic systems, which are continuously evolving.

This section is followed by a discussion of related work. The section that follows presents an overview of G-RTI. This is followed by a description of its architecture and implementation. Results from a benchmarking study are presented both in terms of communication performance and energy efficiency. A case study of a vehicle mounted sensor network emulation using G-RTI is presented. The chapter closes with concluding remarks, and directions for future research.

4.1 Related Work

Many middleware approaches have been proposed in the past. Some support specific types of scenarios and some are more generic (Yu, Krishnamachari et al. 2004, Aberer, Hauswirth et al. 2006). Middleware approaches such as (Aberer, Hauswirth et al. 2006) allow reconfiguration of nodes in a live system but require an XML interface to interface the nodes, restricting dynamic changes in the computational resources of the nodes. These middleware approaches assume sensors are very resource constrained; the approach proposed here assumes a mix of devices with varying degrees of resources available to them. It is up to the system developer to assign computational tasks depending on resource availability, which can be static or vary dynamically.

Approaches such as (Cugola, Jacobsen et al. 2002, Soini, Van Greunen et al. 2007) present middleware that are based on a publish/subscribe method for communication rather than pull-based messaging. (Heinzelman, Murphy et al. 2004, Yu, Krishnamachari et al. 2004) proposed a peer-to-peer middleware approach for sensor

networks. (Soini, Van Greunen et al. 2007) investigates and argues the advantages of a centralized approach over peer-to-peer middleware for sensor networks, citing reasons such as faster synchronization, discovery, routing, and data dissemination. Our experiences thus far with both centralized and peer-to-peer middleware are consistent with these comments. In addition, the centralized approach is better able to support thin-as-required clients compared to peer-to-peer approaches. Another advantage of a centralized middleware is it is more natural in supporting the integration of on-server applications.

4.2 G-RTI Overview

A runtime infrastructure (RTI) is middleware that implements services to interconnect sensors, databases, simulations and other elements in a distributed computing environment. This term is often used in the context of distributed simulation systems such as those realized using the High Level Architecture for Modeling and Simulation (IEEE standard 1516). Systems such as these are intended to support as-fast-as-possible and real-time distributed simulations that combine simulated and hardware components.

The Green Runtime Infrastructure (G-RTI) middleware was designed to be applicable to different contexts and applications. It supports interconnecting simulations and operational devices and software through a set of services for data exchange, time synchronization, and system management. G-RTI supports interoperability among a wide variety of platforms, operating systems, and programming languages that are interconnected through different mediums, e.g., private as well as public wireless and

wired networks. The middleware was designed to be scalable to accommodate many such devices and provide all the functionalities that are expected by distributed simulation and other computation components to execute and exchange data. G-RTI was designed to reduce the effort required to develop a DDDAS, necessitating that application development be straightforward. Specifically, G-RTI utilizes web services and standard protocols to leverage existing software development tools.

Both push and pull based messaging are supported. In push-based messaging the receiver need not be aware of the sources producing messages nor when data will be produced. It is suitable for event based messaging, e.g. a triggered sensor sending data to a service (to save energy) or a simulator listening for events. This kind of messaging is also referred to as publish/subscribe communications. (Cugola, Jacobsen et al. 2002, Souto, Guimarães et al. 2004) argue the importance of this type of messaging to support mobile and sensor networks in general. By contrast, in pull based messaging the receiver actively queries the sender for the message. This is suitable for scenarios with on demand message requirements, e.g., a client requesting data from a continuously operating sensor or a database. Use of G-RTI as a middleware for a wireless sensor network controlled by a centralized routing protocol such as Base Station Controlled Dynamic Clustering Protocol (Muruganathan, Ma et al. 2005) to conserve energy of the sensors thereby maximizing the lifetime of the sensor network is an example of such a system.

An important aspect of G-RTI that distinguishes it from other RTI implementations is its emphasis on reducing the energy needed to interconnect mobile devices, an obvious requirement for many wireless sensor networks. Such systems are generally composed of sensors which are battery operated and are generally not

accessible after they are installed (Schwiebert, Gupta et al. 2002). The energy constraint determines the operation time of each of these devices. It these scenarios, it is beneficial if the middleware itself considers the energy constrained nature of the system and seamlessly makes the system energy efficient without any specific intervention of the system developer(s). An important consideration is to allow for the use of G-RTI as a platform for development and implementation of such energy conservation strategies. This is discussed in greater detail later.

4.3 Design Approach

G-RTI uses a client-server architecture. In the following it is assumed G-RTI executes at a computational node called the *G-RTI server*. G-RTI is not restricted to requiring a central server, however, this simplifies the discussion. A multi-layered server implementation may be used to meet scalability requirements. Extending the client server paradigm, each of the computational nodes interacting with G-RTI, be it a data source (e.g. sensor or databases), a distributed simulation entity or other element is referred to as a *client*. The functionalities implemented by the G-RTI server are referred to as *services*. The functionalities implemented at the client are referred to as *callbacks*. Finally, a client storing a data object is called the *owner* of the data object and all other clients are referred to as *non-owners* with respect to that data object.

As with any RTI there are some rules that guide the implementation and use of the RTI. Due to the generic nature of the RTI the only three rules that applies to the current implementation of the RTI.

1) First, unlike the HLA, non-owners can update data objects. This has two major implications:
    i) This allows for a inclusion of clients such as sensors that have limited computation and storage capabilities. The data objects reflecting the state of the physical object sensed by such a sensor could be stored in a remote client.
    ii) This allows for multiple components of the system to operate on a shared data object.
2) Second, the ownership of data objects can be transferred among the clients. This allows for migration of data among the clients and hence for a change in ownership where the hand-offs are seamless and invisible to other clients.
3) Third, *Thin-as-required* approach. This philosophy is in line with G-RTI's goal to implicitly allow for development and inclusion of clients that can vary in-terms of their constraints regarding computation and energy constraints. This allows the clients to individually (independent of other clients in the system), choose to be as computationally and storage intensive it can support/require. In other words, a client can choose how thin it wants to be.

4.4 G-RTI Services

There are at present three major classes of services provided by the G-RTI: Management, Data Exchange and Time Management Services.

4.4.1 Management Services

Management Services inform G-RTI about the components of the system.

1) *Join*: This service, as the name suggests, notifies the G-RTI server of the arrival of a new client in the system.
2) *Leave*: This service notifies the G-RTI server of the departure of a client.
3) *Register Object*: This service allows the client to register their data objects with the G-RTI server. Registration of objects allows other entities in the system to discover and interact with the data object.

4.4.2 Data Exchange Services

Data exchange services, as the name suggests, allow for exchange of data among the participating system entities.

1) *Update Object Value*: This service allows a client to notify the owner of a data object of an updated value for the object.
2) *Query Object Value*: A client can request/read the value of an object owned by some other client. This falls under the Pull Based data exchange mechanism.
3) *Subscribe Object Value*: G-RTI also allows for a subscription based data exchange mechanism, which allows for the client to be notified of any updates to the subscribed data object. This type of data exchange, as mentioned earlier, is referred to as Push based data exchange. This service allows clients to subscribe to data objects.
4) *Notify Object*: As part of the push based data exchange, this service allows the owner of a data object to notify G-RTI of updates to the object.
5) *Query Reply*: This service allows the client to reply to a read object value request.

4.4.3 Time management services

These services allow G-RTI to support synchronization of simulation (or logical) time among distributed simulations and emulations in the system.

1) *Time Advance Request*: This service is suited for simulations that are time Stepped simulations. It allows the simulation to synchronize their logical time with other simulations in the system.
2) *Next Message Request*: This service is preferable for Event Driven to request for the next available message for the simulation, which implicitly synchronizes the client with other simulations in the system.
3) *Time Advance Grant*: This is essentially a call back from the G-RTI to simulating clients notifying them that it is safe for them to advance to the requested logical time.

4.5 Call back Functions

There are four call back functions that are implemented by clients:

1) *Read Object Value*: This callback is initiated by G-RTI as a response for a query object value. G-RTI expects a value in return (the form of a query reply), so the client requesting the value could be replied to.
2) *Reply*: This callback is initiated when the object owner has answered the client's query.
3) *Revise Object Value*: This call back is initiated by G-RTI as a response to an update object request, towards the owner of the object. A client requires this callback only if it owns a data object.

4) *Reflect*: G-RTI initiates this callback for all the subscribers of a data object, whenever there is a notification of update to the data object. A client requires this callback only if it is subscribed to a data object.

In keeping with the philosophy of allowing the clients to be as thin-as-required, these call back functions are not required, depending on the functionality of the client. For example, a sensor or a simulator, which is only sending data to other clients, will not need any of these call back functions. On the other hand, a data base client might implement all of these callbacks with cases specific to each object it owns or is interested in receiving.

4.6 Usage Scenario

To illustrate typical usage of these services, consider the system depicted in Figure 4-1. As discussed earlier, G-RTI supports push and pull based messaging. Figure 4-1 shows a push based messaging scenario. In this scenario, client 1 is a vehicle mounted mobile sensor relaying real-time location and velocity information while client 2 is one of several aerial drones in a swarm responsible for monitoring traffic. Client 3 includes a database that records updates and executes a predictive simulation using historical and online data. Finally, *x* denoted an object containing vehicle information that is generated sporadically in an unpredictable fashion that is of interest to clients 2 & 3.

All clients join the system (asynchronously). Client 3 owns the data object and maintains its state. It informs G-RTI of its ownership of object *x* using the Register service. Client 2 is interested in updates to object *x* but does not own the object, so it

subscribes to any updates to *x*. Finally, client 1 is a source of generation of data object *x*. So, whenever client 1 wants to convey a new value for *x*, it uses the Update service to inform the G-RTI of an update. G-RTI then forwards the update to the appropriate client (client 3). In addition, G-RTI also reflects the updated value to any subscriber (client 1).

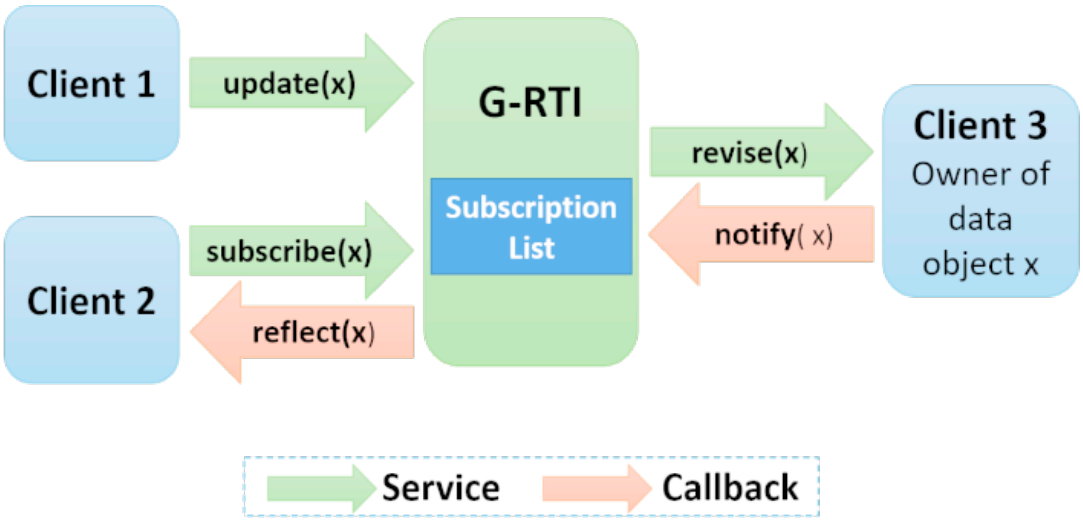

Figure 4-1: Push based message service usage scenario.

Another scenario illustrating the push mechanism arises when the owner of the object itself generates an update for a data object. In the example scenario, the predictive simulation might want to fill in for any missing update. Using the previous mechanism that uses the update service of G-RTI would trigger a revise callback, which is redundant and might require extra effort on the application developer of client 3 to remove a possible update loop. G-RTI's notify service handles these cases. Client 3 notifies G-RTI of an update for object *x* and G-RTI reflects the update to all the subscribed clients.

We extend this scenario to illustrate use of pull based messaging. Client 2 would like to know the value of x. It queries G-RTI for this value, which then relays a read

request to the owner of the object (client 3). The owner replies to G-RTI using the query_reply service. G-RTI then relays the reply to the requesting client. All these steps are executed asynchronously, which allows flexibility in the application development.

It might be noted that client 1 in this example may be resource constrained. The client only requires resources to sense/gather the required data and to make an `http` request. Client 2 may be able to support more extensive computation but might be constrained by a limited amount of storage. Client 3 might be executing on a server with extensive computation and storage resources. Allowing the clients to be thin-as-required independent of other clients allows them to co-exist and form symbiotic relationships with other clients.

4.7 Implementation

Design choices in implementing the system were governed by the goals discussed earlier. Web interfaces were used to place minimal requirements and restrictions on clients and to simplify client-side application development. All services can be accessed using a URI, hence allowing for any device capable of connecting to the Internet to become a client. Apache was used because of its proven ability to scale to large deployments. CPPCMS is a C++ based framework that allows a native, high performance interface for inclusion of on-server applications. It was used to provide high performance and scalability. Popular packages and simulation applications used for development and study of networked and distributed systems are based on C++. Finally, Apache and CPPCMS are both free and open-source software platforms, simplifying widespread distribution of G-RTI. Apache is released under Apache License 2.0 and CPPCMS with

an LGPLv3 license and alternative Commercial License as an alternative for proprietary software development.

Push based messaging is a not a native operation for both socket and web based network communications. G-RTI implements push based messaging using continuous polling. Simply put, continuous polling is maintaining at least an instance of client request for each client in G-RTI. As can be easily seen, with finite resource there will be a time the server runs out of instants for a client. The state for the client must be reset to polling before any further messages can be pushed to the client. We call this period the resetting period and discuss it further in the benchmarking section.

Another feature of G-RTI is a dynamic web-dashboard. The dashboard allows remote monitoring of the system components and eases system development and debugging. Currently the dash board provides two views, a client level view that list all the information pertaining to a client, and a data object level view that presents information organized by objects.

4.8 Benchmarking Experiments

We next present results from micro-benchmarking experiments to evaluate the performance of the G-RTI services. This study focused on two major performance metrics of the middleware implementation. First are the latencies of the messaging primitives provided by G-RTI. The second concerns the maximum bandwidth of communications for each G-RTI client to update a shared object.

The setup for the benchmarking experiments consisted of an implementation of G-RTI executing on a Lenovo ThinkPad T410s laptop with an Ubuntu 14.04 LTS operating system. The setup also included two (three in case of subscription latency) python clients, on the same system. The choice was made to avoid time synchronization issues for computing subscription latency. All other results were replicated on the same platform to maintain uniformity for comparison purposes. All the experimental values presented represent the average of 100 trials.

We define subscription latency as the time between the instants when an update is sent until a subscribed client receives the update. The subscription latency was computed by using three clients in a scenario as shown in Figure 4-1. The implementation follows the first push based message scenario described earlier. Briefly, client 1 sends an update to an object owned by client 3 and client 2 is subscribed to the object and hence receives a reflect call back. The subscription latency is computed as the difference in the time instant when the update request to the object starts in client 1 and the instant when client 2 successfully completes receiving the reflect callback. As the clients are on the same system the time difference is not affected by any clock synchronization. This is then repeated for varying sizes of update messages.

We define the query round trip latency as the minimum time from when a client sends out a query for an object until the time it receives a reply from the G-RTI server. As noted earlier, query is an asynchronous operation and hence the query latency can vary depending on how the clients are set up. For computing the query latency, the experimental setup and the implementation is the same as described in the scenario for pull based messaging discussed earlier. Briefly, client 2 queries G-RTI for the value of

the object owned by client 3, which then receives a read callback from G-RTI and replies back to the query. Finally, client 2 receives a reply from G-RTI. The latency is computed as the time difference between the instants when client 2 starts sending the query and the instant when it finishes receiving a reply. This is repeated for varying sizes of object values.

We define update round trip latency as the time between the instant when a client sends an update to a peer client and the instant when the client receives an update from the peer client. To compute the latency, client 1 sends an update to an object owned by client 2 and vice-versa. The round-trip includes an instance of the resetting period. The resetting period was empirically determined to be approximately 4 milliseconds. Again, the update length was varied.

Figure 4-2 shows the results of the latency benchmarking experiments. The subscription latency is the least, and this can be attributed to the fact that it requires one-way communication. Both query and update round-trip times include one resetting period, which is a constant irrespective of the message size. Furthermore, the change in size of the message affects only the reply path of the query latency, i.e., the size of the reply changes. The other path is not affected as the size of the query requests stay constant. This explains a lower slope of the line for the query latency with respect to the update round-trip latency and a comparable slope for subscription latency. Finally, the difference in the slopes for update-round trip latency and the query latency can be easily accounted for when the resetting period is considered as a constant in both. The constant slopes signify a linear increase in latency as the message size is varied. These results show a very predictable behavior for G-RTI with respect to the latency for the messaging

services. Because the subscriptions, objects and clients are managed as hash maps, which provides a constant time look up, an increase in the number of any of these is not expected to affect the latencies.

The maximum bandwidth excluding http and lower layer headers was measured to be approximately 3.7 Mbps (and 2.21 Mbps when the resetting period is considered). These were computed without any manipulation of the default settings of the Apache web-server, so should be considered to be conservative values. The bandwidth is currently limited by the size of the request that Apache and FASTCGI allow. Theoretically the request size can be set to any arbitrary number. In that case the limiting factor is the network hardware and software stack. Even so, the bandwidth is comparable with other web-based RTIs (Möller and Dahlin 2006).

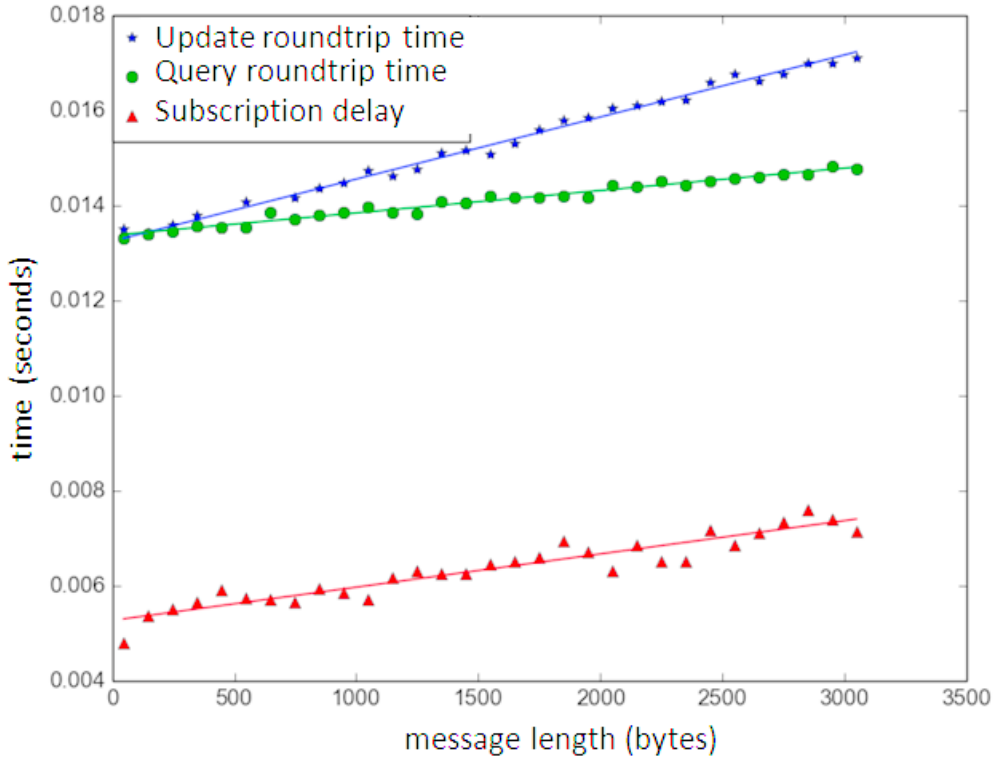

Figure 4-2: G-RTI messaging primitive latencies.

The throughput and latency are implementation dependent, making direct comparison difficult. However, widely used micro benchmarks are reported in the literature. E.g. one can compare the results presented here with those presented in (Cardoso 2017). G-RTI performs significantly better than the presented middleware implementations on all counts.

4.9 Energy Consumption

An approach of reducing energy consumption is message aggregation. If the simulation must send a stream of update messages to the server, one could aggregate several messages into a single message, and send one larger message rather than a sequence of smaller messages. This approach, termed *message aggregation*, is commonly used in parallel and distributed systems in order to reduce communication overheads. Message aggregation has been used to improve the efficiency of communications in both loosely coupled distributed systems such as sensor networks (Naldi and Willig 2008), Vehicular Ad-Hoc Networks (VANETs) (Raya, Aziz et al. 2006) and mobile computing environments (Lau, Cherukuri et al. 2013) and tightly coupled systems such as shared memory high performance computing (HPC) systems (Pham and Albrecht 1999). Answers to the questions "what messages" and "how many messages" should be aggregated are very closely tied to the application and determine the impact and the efficiency of the approach. For example, work presented in (Saleet and Basir 2007) describes a location based aggregation protocol for aggregating messages originating/destined for geographically co-located vehicles in VANETs. Message

aggregation comes at the cost of increasing latency, as some messages must be held at the sender in a buffer while the data is being accumulated, rather than immediately sending the data to the receiver. Work presented in (Khanna, Naor et al. 2002), looks at the latency and performance tradeoffs for control messages in a multicast environment.

To study the effect of the aggregating messages on the energy consumption of the client we compute the energy consumed for varying level of aggregation. The experimental study was conducted using G-RTI. For this study, which formed a part of a larger study presented in (Hunter, Biswas et al. 2018), a cellular automata traffic simulation based on (Rickert, Nagel et al. 1996) was developed and configured to model the NGSIM section of Peachtree Street in midtown Atlanta. This simulation was executed on the client, a cellular phone.

The experimental setup consisted of an Android® smartphone (Google Nexus® 5), running Android 5.1 as the client, and a Lenovo ThinkPad® running Ubuntu 14.04, as the G-RTI server. All the communications in the experiments were conducted using 802.11n Wi-Fi over public Wi-Fi access points. The client runs a multithreaded Native Android application with the simulation developed in C and communication using Volley, an HTTP Android library. The Trepn® application (Qualcomm_Technologies 2015) developed by Qualcomm was used to measure energy and power consumption of the client machine.

In each experimental run, the number of updates, and the size of the data in each update remain constant and the only variable is the number of updates aggregated to form a message. In these experiments the application executed the cellular automata based

traffic simulation; and in addition the application also aggregated and sends messages. The amount of energy consumed per byte of transmitted data was measured. Figure 4-3 shows the results of this experiment. As expected, an initial reduction in the energy consumed is observed as message aggregation increases. This is because fewer messages are sent, thereby reducing the energy consumed to process the message and send overhead information such as message headers. However, an inflection point is reached, and the energy consumed per byte of data starts increasing beyond a certain level of aggregation. Also, from the standpoint of energy consumption, data aggregation is only effective up to the maximum data packet size used by the underlying operating system.

As expected, an initial reduction in the energy consumed is observed as message aggregation increases. Thus, from the standpoint of energy consumption, a rule of thumb could be that data aggregation is very effective. It shall be noted that depending on the message aggregation strategy used, it can introduce latency in message delivery.

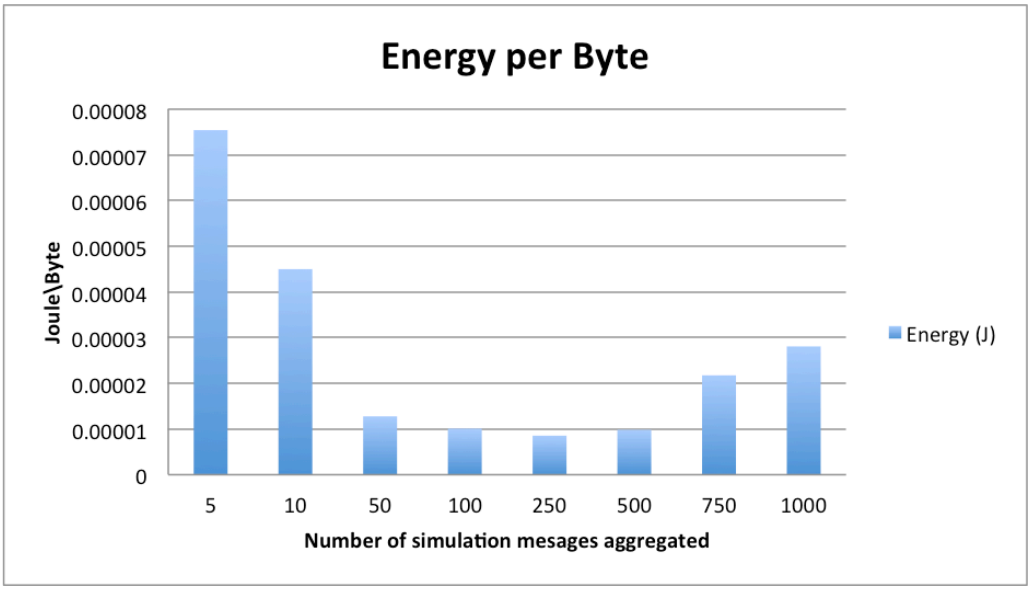

Figure 4-3: Energy reduction by clustering messages.

4.10 Case Study: DDDAS using G-RTI

Crowd sourcing, mobile computing and a wide variety of public sensor based intelligent/smart systems have made a many type of data available for transportation systems. Combined with the emergence of autonomous vehicles and smart cities (Shueh 2017), intelligent and feedback based systems for transportation system operations and management are gaining in importance. A recent development in the field is the standardization of a message set for the DSRC (Dedicated Short Range Communication) to provide interoperability among DSRC applications (SAE 2009). DSRC as defined (Sill 2016) by the United States Department of Transportation (DoT), is a two-way short- to-medium-range wireless communications capability that permits very high data transmission critical in communications-based active safety applications. One of the messages defined in the message set dictionary is the basic safety message (BSM).

Applications such as that proposed in (Synesis_Partners_LLC 2013) derive weather information from mobile sensors on the vehicles transmitted as part of the BSM message to assist the management of the roads under adverse weather conditions. Also, the vehicle operation information present in Part A of the BSM message can be utilized for user applications such as vehicle emission modeling proposed in (Guensler, Liu et al. 2017) and energy efficient route prediction. These applications use the vehicle-mounted sensors as a network of mobile sensors to generate an understanding of a larger system, which is then used to provide added benefits to the users. Similar DDDAS based traffic

systems were studied in (Fujimoto, Guensler et al. 2006, Hunter, Fujimoto et al. 2006, Hunter, Kim et al. 2009).

Development of these applications is not straightforward. Because they rely on the BSM message to capture and predict future states of the system, it is important to understand and consider the uncertainties of mobile wireless communications. Packet drops, e.g., due to network congestion, can greatly affect the services built on top of such networks. Early evaluation and development coupled with the need for reproducible results necessitate the use of simulations operating in controlled environments such as the developer's laboratory. Simulation models of vehicle traffic and wireless communication are needed, but detailed, high resolution models of both are seldom available within the same simulation package, calling for an integrated approach. And to natively test software applications the developers must also interface the application with the set of simulators, which can be time consuming. This is further complicated if the application needs to be studied in a real-time environment where an emulation capability is needed. Often these efforts are specific to the study at hand and cannot be reused.

Figure 4-4 depicts a DDDAS application for transportation system management. In this system a traveler assistance app is depicted that executes on a smartphone. It utilizes updates from a back-end server application. The server application relies on BSM data communicated among a network of mobile sensors mounted on the vehicles augmented with other inputs such as weather data to deliver information relevant to the app's user, e.g., data such as travel time and fuel consumption concerning the driver's planned route as well as that for alternate routes.

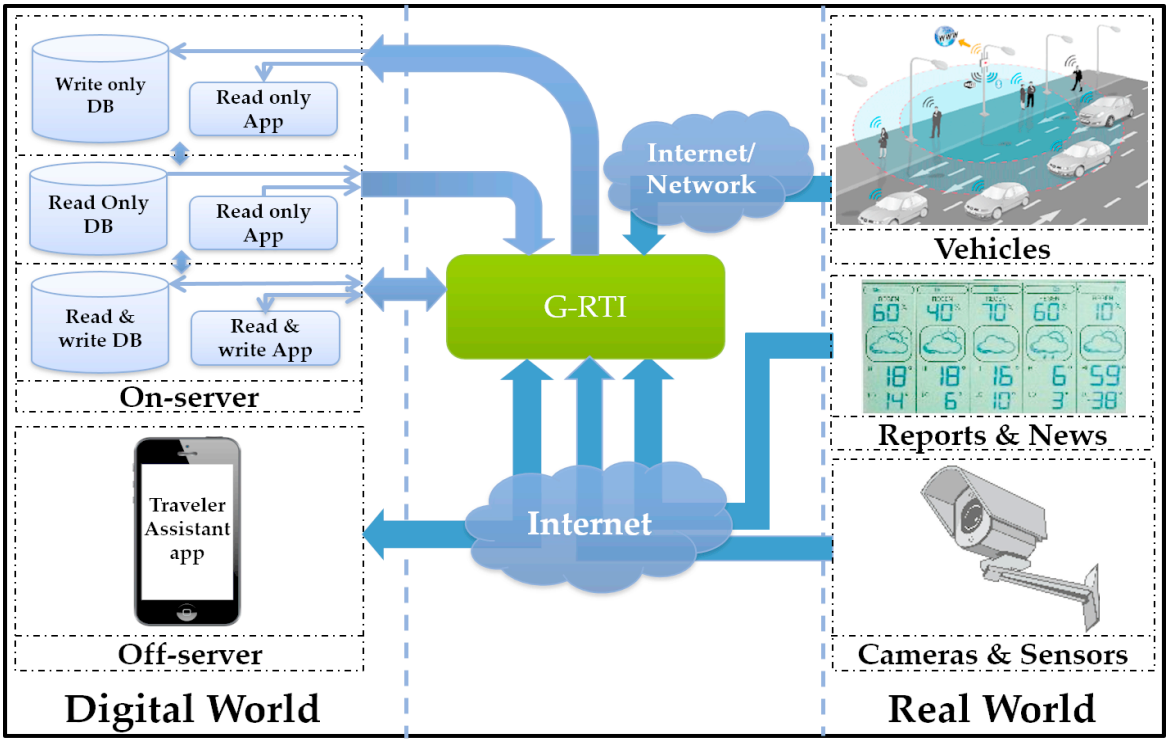

Figure 4-4: Notional diagram of DDDAS for case study.

(V2X image: © Zoloo777) CC BY-SA 4.0 license.

In this study we simulate/emulate a road network with connected vehicles, communicating BSM messages. In addition to showing an end-to-end implementation using GRTI, this study also provides a close to real-world example of how G-RTI can be used to integrate different simulations. The traffic simulator models the mobility of the vehicles. A communication network simulator models information exchanged between vehicles and roadside base stations. Here, we utilize two widely used simulators, SUMO (SUMO 2005) a traffic simulator suite and NS3 (NS-3_project 2011) a network simulator. The road network (SUMO_application_network 2005) has a total edge length of 17.84 Km and a total lane length of 49.53 Km. The network has four origins, four

destinations and two intersections. Each road for outbound traffic has three lanes. U-turns are prohibited at all intersections. A higher priority has been given to eastbound and westbound traffic. Vehicle data is output in real-time in floating car data format. The format includes the location of all the vehicles in the simulation in addition to other vehicle data. The simulation using NS3 models moving nodes (vehicles) on a "dynamically specified" path, communicating BSM messages over WAVE. This can be broken down to two major parts, first the mobility simulation for the nodes in NS3 and second the BSM communication simulation.

The mobility for the nodes in NS3 was implemented using NS3's waypoint mobility model augmented with support for dynamic inclusion and removal of vehicle during the simulation. The waypoint mobility model requires the position of the vehicle and the time-stamp at which the vehicle's position was recorded (in this case provided by traffic simulation using SUMO). The waypoint mobility model can be thought of as a list of vehicle position and timestamp data. Hence it requires at least one time-step into the future. This was achieved by simulating SUMO a time-step ahead of NS3 (Eichler, Ostermaier et al. 2005). Another constrain of NS3 is that it does not allow dynamic creation of nodes. The dynamic inclusion and removal were made possible by recycling of NS3 nodes and assuming the number of vehicles in the simulation at an instant has an upper bound. This was facilitated by the *max-num-vehicles* argument of SUMO. Another part of the implementation was that the requirement of dynamic paths for the nodes required the vehicle trace to stream to NS3 in real-time from SUMO. This was achieved by using TAP device, which is exposed on one end as a kernel net device and on the

other end as a file descriptor in user space. The user space file descriptor can then be passed on to `FdNetDevice` of NS3.

The next part is to model BSM communication for the nodes. The BSM communication was simulated with a packet size of 200 bytes (Sung, Noh et al. 2013) with a frequency of 1 Hz with an expected range of 1000m (Cronin 2012) without channel switching and with a max random delay of 10ms before transmitting over WAVE. The simulation was implemented using the `WaveBSMhelper` application class of NS3. Figure 4-5 shows the effect of increasing the number of nodes in NS3; the vertical axis presents the wall clock time taken by NS3 to simulate one simulation second. The quadratic behavior of the curve can be explained by the nature of the broadcast. In this scenario the limiting factor for achieving real-time performance is the execution time of NS3. This system was developed and tested on Ubuntu 14.04 LTS. It must be noted that the simulations can be executing on other Operating systems as well e.g. VISSIM a proprietary windows based simulator could be used to replace SUMO in this study. A view of the simulated road network is shown in Figure 4-6. Traces for the vehicles on the emulated road network were generated by SUMO. They are parsed by a python script and passed on to another python script through G-RTI, which then relays it to NS3 through the FD-TAP interface. Figure 4-7 illustrates one frame in an animation of the mobility simulation and network simulation outputs. Although the simulations are run in real-time and concurrently, the NS3 visualization was captured offline due to the lack of a good online visualization program for NS3.

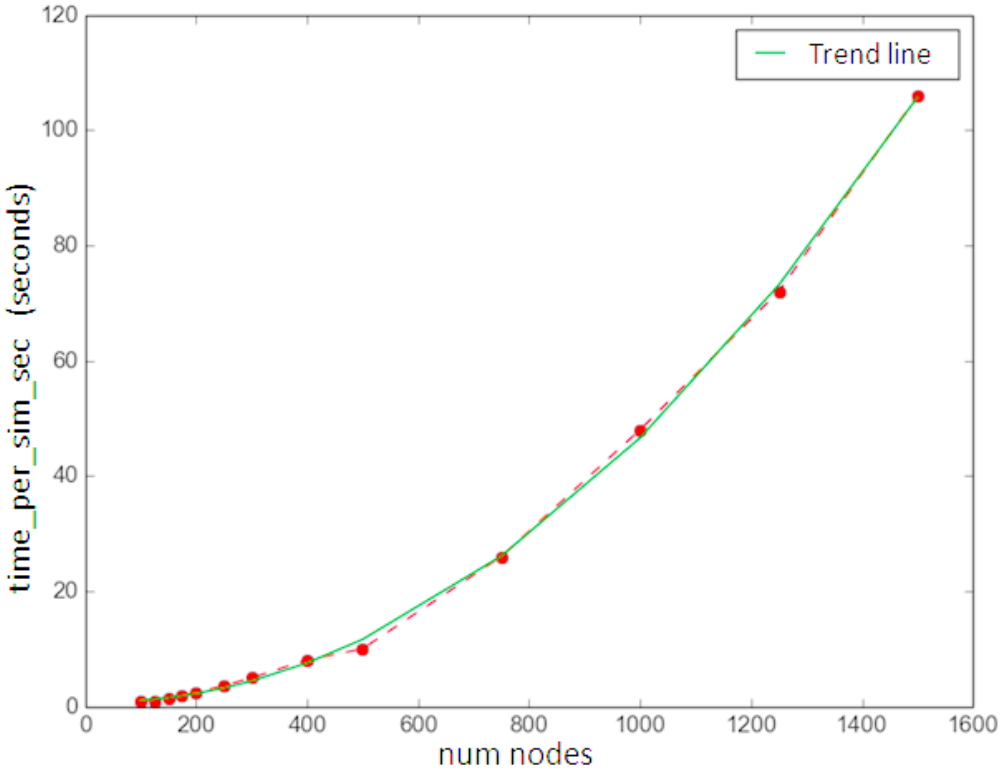

Figure 4-5: Effect of changing number of Nodes in NS3 on wallclock time versus simulation time.

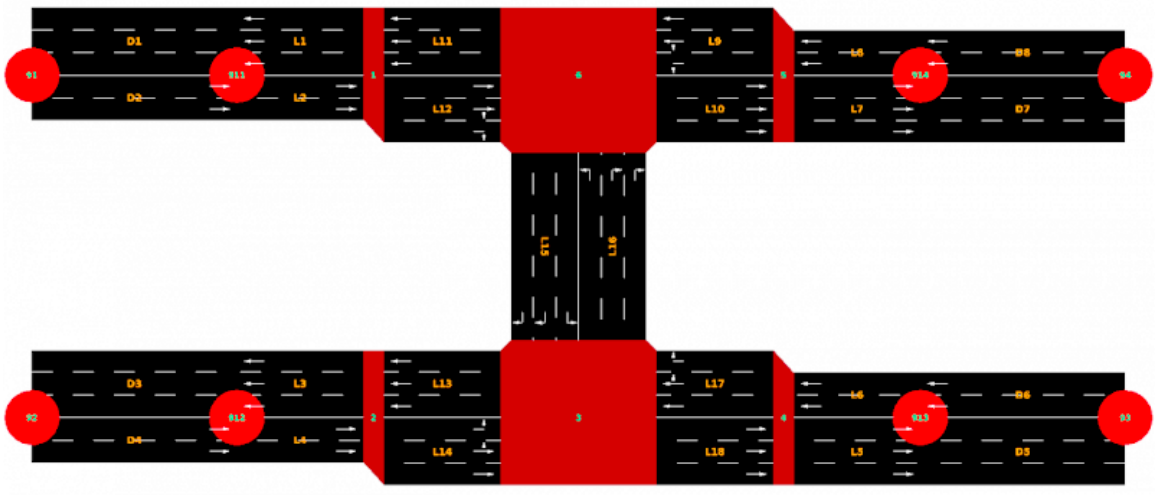

Figure 4-6: View of simulated road network (not drawn to scale).

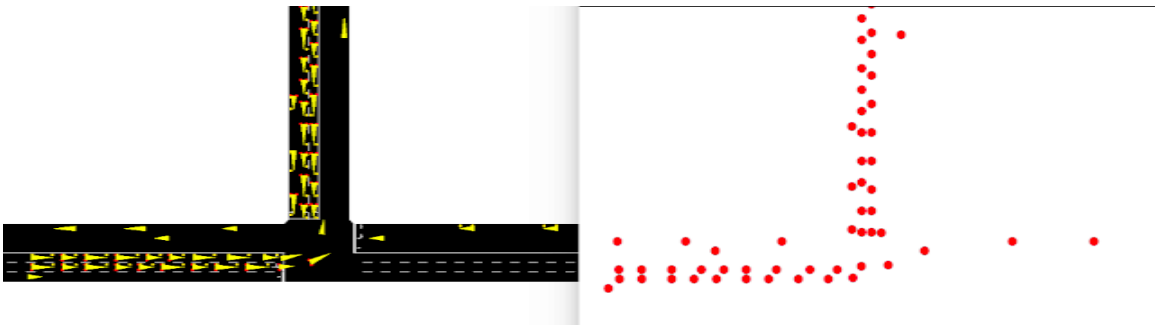

Figure 4-7: Graphical outputs of SUMO (left) and NS3 (right) at same simulation time.

4.11 Conclusion

This chapter described G-RTI, a middleware to support DDDAS developments consisting of heterogeneous components. G-RTI provides a common platform for research, emulation and deployment of systems. An important goal for G-RTI is to improve energy efficient transparent to the application. In addition, major design goals include support of a wide variety of devices and application contexts, reduced development effort, providing a platform for energy efficient deployment and the possibility of thin-as-required clients. The suite of services offered by G-RTI was developed from our experience in the design and implementation of distributed systems ranging from purely analytical system to systems with real-world components. We consider G-RTI as toolbox where the system developer has the choice of using the tools that they find most efficient for the specific system development.

The services offered by G-RTI and the expected callbacks were described as well as implementation details and usage scenarios. Performance measurements from a benchmarking study characterize the delays required for various messaging services and some energy consumption properties.

5 ENERGY EFFICIENT SYNCHRONIZATION OF DISTRIBUTED SIMULATIONS

Previous chapters discussed the increasing importance of the energy and power consumed by computing applications. It was also pointed out that despite these trends, relatively little work has been completed to date concerning the energy and power requirements of parallel and distributed simulation codes. Building on the observations presented in chapter 3, this chapter focuses on the subject of reducing the power and energy consumption properties of key algorithms. This chapter introduces a new synchronization algorithm to improve energy efficiency transparent to the application, an important goal for G-RTI.

Distributed simulation programs commonly assume the computation consists of a collection of logical processes (LPs) that exchange timestamped events, or messages. A synchronization algorithm is required to ensure that the parallel execution produces the same results a sequential execution where all events are processed in timestamp order. Synchronization algorithms are commonly classified as conservative and optimistic methods. Conservative approaches block LPs to ensure no LP ever processes events out of timestamp order. Well known examples include the Chandy/Misra/Bryant (Chandy and Misra 1981) and YAWNS (Nicol 1993) algorithms, among others. Optimistic algorithms use a detection and recovery approach where synchronization errors, i.e., out of order event executions are detected during the execution, and their effects are erased through a rollback mechanism. Time Warp is the most well-known optimistic synchronization algorithm (Jefferson 1985).

In this chapter we first review related work in power and energy consumption for parallel and distributed simulation codes. The minimum energy requirements for a distributed simulation program are defined. A property called *zero energy synchronization* is introduced for synchronization algorithms, and a theoretical approach to achieving zero energy synchronization is discussed. An energy-optimized implementation of YAWNS, termed Low Energy YAWNS (LEY) is described as a practical approach to reducing the energy consumed for conservative synchronization. It is shown that LEY can, in principle, achieve zero energy synchronization for a large class of distributed simulation applications. This is followed by the results of an experimental study. This study measures the energy consumed by LEY for a set of benchmark applications. To empirically portray the advantages of LEY, the energy consumption of LEY is then compared with the energy consumption of YAWNS and a zero energy synchronization algorithm for a set of benchmark application. This is followed by a discussion of future work and conclusions.

5.1 Related Work

There is an extensive literature in power and energy consumption and optimization techniques in the mobile and embedded computing literature, and a growing body of work considering such issues in high performance computing. However, as discussed below, there is only a limited amount of work on power and energy consumption of distributed simulation programs.

The bulk of the work examining energy and power consumption in areas such as mobile computing and embedded systems focuses on low-level elements such as

hardware, operating systems, and compilers. For example, much work in operating systems considers implementation techniques to reduce power consumption while meeting performance requirements and deadlines (Quan and Hu 2001, Saewong and Rajkumar 2003, Cho, Liang et al. 2011). Many ad hoc communication protocols have been designed for low power operation. Power mode management techniques exploit low-power modes of operation for processors, memory, storage, and communication circuits (Niu and Quan 2004, Hoeller, Wanner et al. 2006, Bhatti, Belleudy et al. 2010), e.g., disabling components or switched them to power saving states. A substantial body of work focuses on utilization of dynamic voltage and frequency scaling (DVFS) where voltage and clock frequency can be reduced to trade off power consumption with execution time (Hua and Qu 2003, Ge, Feng et al. 2005, Freeh, Lowenthal et al. 2007).

Work in energy and power consumption for parallel and distributed simulation is surveyed in (Fujimoto 2017). Empirical studies comparing the power consumed by conservative synchronization algorithms are presented in (Fujimoto and Biswas 2015, Biswas and Fujimoto 2016, Biswas and Fujimoto 2017). Measurements of energy consumption in a Time Warp system are described in (Maqbool, Naqvi et al. 2017). Use of dynamic voltage and frequency scaling to optimize Time Warp programs is described in (Child and Wilsey 2012). A study comparing the power consumed for cellular automata and queueing network models for vehicle traffic in distributed simulations is presented in (Neal, Fujimoto et al. 2016).

5.2 Minimum Energy

We begin this discussion by defining the minimum amount of energy that must be consumed to execute a distributed simulation program. We assume the program consists of a collections of N logical processes LP_1, LP_2, … LP_N and the computation performed by each LP is a sequence of event computations, where $E_{i,j}$ denotes the j^{th} event executed by LP_i. Let $C_{i,j}$ denote the minimum energy consumed in processing event $E_{i,j}$, exclusive of the energy required to schedule new events, discussed next. The simulation is initialized with some number of scheduled events; the energy required to initialize the simulation is not considered here. Each new event created during the simulation comes about through an event scheduling operation. Let $S_{i,j}$ denote the minimum energy required to schedule $E_{i,j}$. If the new event is scheduled on a different processor from the LP scheduling the event, $S_{i,j}$ is defined as the energy required to send and receive the message containing the event; we implicitly consider such communications as part of the distributed simulation computation itself. If the sender and receiver reside on the same processor, we define $S_{i,j}$ to be zero. This is a simplification because some energy must be expended to allocate storage and place the event into data structures such as the local event list, however we ignore this energy to keep the model simple; inclusion of this energy, e.g., as a fixed energy cost, is a relatively straightforward extension to the model. We assume the program continues to run until there are no more scheduled events to process. We define the minimum energy required to execute the distributed simulation program as simply the energy required to process and schedule the events that it processes:

$$MinEnergy = \sum_{i,j} C_{i,j} + S_{i,j}$$

We note that this formulation does not include the energy required for the distributed simulation synchronization algorithm. For example, it does not consider energy consumption for null messages or global synchronizations in conservative algorithms and considers only committed events for optimistic synchronization algorithms. We characterize such operations as overhead, with the goal that developers of distributed simulation systems will strive to minimize this overhead, subject to other performance goals such as minimizing execution time. We note that this formulation does depend on the application configuration[5], i.e. mapping of LPs to processors and the resulting communication pattern. This formulation also excludes energy for I/O operations. However, such can be easily added if significant or important for a particular simulation. Finally, it is assumed no energy is expended when a processor is idle, i.e., it only considers dynamic power consumption.

5.3 Zero Energy Synchronization

The above definition for *MinEnergy* is intended to separate the energy required for distributed simulation computations, e.g., producing new computational results and distributing information to other LPs, and that required for synchronization. The expenditure of energy to produce such results seems reasonable. However, the need to expend energy to synchronize computations is less clear. In a distributed simulation

[5] Much like speed up of a parallel (or distributed) algorithm is dependent on the performance of the baseline sequential algorithm, the choice of implementation details of the baseline system while computing *MinEnergy* will affect any comparison made. We implicitly imply that the best possible setup is used, and leave out such implementation details for the simplicity of the model and its discussion.

program, the synchronization algorithm does not perform computations that directly contribute to computational results. Rather, the purpose of the synchronization algorithm is to ensure that event computations are performed in a proper sequence to produce the same results as the corresponding sequential execution. As such, we exclude the energy required for synchronization from the definition of the minimum energy required to complete the distributed simulation computation in order to provide a basis for minimizing the energy needed for synchronization.

The above discussion raises the question of whether one must expend energy to synchronize a distributed simulation. Is it possible to synchronize a distributed simulation without any expenditure of energy beyond that required for event computations and communications? Are there fundamental requirements to consume energy for synchronization that cannot be avoided? In practice, can one hope to achieve zero energy synchronization, or close to zero energy synchronization? These are some of the questions we begin to explore here.

We define a *zero energy synchronization algorithm* as one that results in executions of a distributed simulation that requires no more than *MinEnergy* energy to complete. We next describe a simple, albeit theoretical, approach to achieving zero-energy synchronization.

Is it possible to achieve zero energy synchronization? Consider a distributed simulation program that does *not* utilize any synchronization algorithm. Each LP simply processes any unprocessed events it has in its future event list (FEL) in timestamp order, and blocks if the FEL is empty. This is equivalent to the execution of a Time Warp

program that does not include operations for state saving, rollback, GVT computations, or other Time Warp specific operations. Consider an execution of this program where it happens that during the execution, the time stamp of each message received by an LP is larger than the timestamp of the last message the LP had processed. It is apparent that the energy required for this execution will be *MinEnergy*. Thus, this execution would require no energy for synchronization, satisfying our requirement for zero-energy synchronization.

Of course, the above approach does not *guarantee* correct synchronization for all executions of the code, only for one particular execution. However, it is suggestive of an approach to achieving zero energy synchronization for arbitrary distributed simulation codes for any execution. If each LP had an oracle to tell it the next event that it should process at any instant in time, then the LP would know whether to wait for this event, or to go ahead and process the next event residing in its local FEL. More precisely, the oracle function is defined as:

$$\mathrm{E}_{i,j} = OF(i,j) \qquad (1)$$

OF returns a unique identifier for the j^{th} event to be executed by LP_i. We assume the *OF* function requires no energy to execute. If LP_i has processed j events, then it simply calls *OF (i,j+1)* to obtain the identifier for the next event it is to process. If the event resides in its local queue, LP_i processes the event. If not, some other LP must generate the event, so LP_i blocks until this event is received. There are several ways to implement such an oracle. For example, an oracle that returns the minimum time stamp

among events the LP will later receive will also similarly allow for zero energy synchronization.

In practice, an implementation of the oracle could be achieved by first obtaining a log of an execution of the distributed simulation code. Alternatively, if one has apriori knowledge of the application, it may be possible to realize *OF* using knowledge of what events are scheduled by each LP rather than using a log. For example, if one knows that communications among LPs follow a ring topology and one can deduce the timestamp of messages it will receive from its neighbors, an implementation of the log using a minimal amount of energy can be achieved. This approach is utilized in the experiments described later.

5.4 Optimizing Energy in YAWNS

We now present an energy-optimized implementation of the YAWNS synchronization protocol called Low Energy YAWNS, or simply LEY.

5.4.1 YAWNS

We first review the YAWNS algorithm discussed in (Nicol 1993). The terminology used here is adopted from that work. Specifically, a simulation is composed of a set of the logical processes. Each simulation LP (or simply LP) simulates a logical collection of entities (or servers[6]). For this initial discussion of YAWNS, we assume each LP consists of a single unit capacity server. A simulation event, defined as the atomic unit of work for the simulation, with a timestamp TS denotes a job arrival, and the server

[6] Note that this server is a logical simulated entity, which is different from the server in the client-server implementation discussed later.

can begin serving/handling the job any time on or after simulation time TS. A unit capacity server is one that can only handle one job at a given simulation time. This implies that simultaneous simulation events corresponding to simultaneous job arrivals will result in the jobs being handled sequentially. An infinite capacity server can be viewed as a dynamic collection of unit capacity servers, which can elastically scale depending on the number of jobs being processed concurrently. This implies that an infinite capacity server can serve any number of jobs at the same simulation time instant.

Define *t* as the minimum timestamp of any event in the entire simulation at one instant during the execution. Define $d_i(t)$ as the lower bound on the earliest completion time of any pending event with timestamps t, on LP_i.

With the assumption of each LP simulating a single unit-capacity server, given t, any LP_i, has a unique value for $d_i(t)$. With these assumptions and definitions, $d_i(t)$ is computed as

1. If the event list of LP_i is empty then, $d_i(t) = \infty$.

2. If LP_i has pending events in its event list, assuming no further events will be inserted into the event list, $d_i(t)$ is set as the completion time of the next event. This is equivalent to the timestamp of the next event the LP may send to another LP if it does not receive any new job arrival events in the future.

The assumption that each LP simulates a single unit capacity server can be relaxed, by computing $d_i(t)$ for an infinite capacity server as the minimum $d_i(t)$ of all

individual servers that make up the infinite capacity server. The same can be extended to a multi-server site.

Next we define the lower bound on time stamp or LBTS as

$$\text{LBTS} = \min_{\text{for all sites } S_i} \{d_i(t)\} \tag{2}$$

LBTS provides a lower bound on the timestamp of any event that can be created in the simulation given t. Along similar lines, $d_i(t)$ can be seen as a local lower bound on time stamp of any event that can be created by LP_i. Hence, we denote $d_i(t)$ as $LBTS_i(t)$. Assuming each LP is composed of a single server, equation (2) reduces to:

$$\text{LBTS} = \min_{\text{for all LP } LP_i} \{\text{LBTS}_i(t)\} \tag{3}$$

Finally, we define the term epoch or window. As will be demonstrated shortly, each LBTS computation serves as a global barrier for synchronization. Hence, we define each update of LBTS as the end of an epoch and start of a new epoch. For simplicity, we define the LBTS value of the n^{th} epoch as $LBTS_n$. Hence the n^{th} epoch is marked by window ($LBTS_{n-1}$, $LBTS_n$].

With these in place, we now describe the algorithms. We assume each LP consists of a future event list (FEL) implemented as a priority queue, where the priority of the event is the timestamp associated with the event. Processing an event may create new events. Each event specifies a destination LP, which can be the generating LP (local event) or a remote LP (remote event). Any received event is enqueued in the FEL of the destination LP.

Algorithm 5-1 describes the YAWNS algorithm based on the assumptions and definitions presented above.

Algorithm 5-1: YAWNS

```
1  LBTS = 0
2  WHILE termination criteria not met
3    IF (FEL not empty && timestamp of FEL.top <= LBTS)THEN
4      process and remove top event
5      communicate generated events to their destination
6    ELSE
7      compute LBTS_i(LBTS)
8      cooperatively compute LBTS
9    END IF
10 END WHILE
```

5.4.2 Low Energy YAWNS

In our prior work (Biswas and Fujimoto 2016) empirical evidence suggested that communication can be a major component of the energy consumed by a distributed simulation, hence providing an avenue for optimization. Further empirical studies, presented in section 4.9 and (Fujimoto, Hunter et al. 2017) on the effect of communication patterns on energy consumed by distributed simulations indicated that message aggregation allows for a large reduction in energy consumption. Finally, we observe that when implemented on client-server machine architecture, additional opportunities for message aggregation could be exploited. Clients can bundle all messages that need to be sent to any other client and send it to the server, and the server takes care of the message delivery to individual clients where the messages can be re-bundled based on destination. We observe that timestamp information necessary to compute LBTS values can be piggybacked onto other messages to reduce the energy consumed for global reduction computations. These observations motivated the development of LEY.

In the following we assume that increasing the number of messages that are aggregated reduces the energy consumed per bit of the total data being communicated. We relax this assumption after describing LEY.

Using similar terminology that was used to describe YAWNS, LP_i of a simulation synchronized with LEY and implemented with a client-server architecture (where each LP is a client) proceeds as presented in algorithm 5-2.

Algorithm 5-3 presents the pseudo code for the LEY server. It should be noted here that a special case might arise if all LPs exhaust all their respective events in an epoch. The LBTS computed by the server would be infinity. In such a case the server can use its omniscience to compute the LBTS. For a unit-capacity LP, this would be the minimum time stamp of any event that would be sent to another LP in the next epoch. This lower bound can be further improved by considering application properties, e.g., exploiting lookahead.

As pointed out earlier, a major source of improvement in the performance of LEY is the grouping of communications by delaying the communication until the end of the epoch. The piggybacking of the LBTS value for each epoch, further reduces the energy cost for synchronization.

Assume that after a message size of m, energy consumed per bit increases. Then the first assumption can be relaxed by introducing a forced synchronization in a client if the aggregated message size (size of the message_buffer in algorithm 5-2) reaches m. This would still keep the synchronization protocol conservative, as this essentially reduces the size of the epoch and might shift the starting time of the following epoch.

Algorithm 5-2: LEY: Client side

```
1  LBTS = 0
2  WHILE termination criteria not met
3    message_buffer = []
4    IF FEL not empty && timestamp of FEL.top <= LBTS) THEN
5      process and remove top event
6      push(generated event, destination) in message_buffer
7    ELSE
8      compute LBTS_i(LBTS)
9      //piggyback LBTS_i(LBTS) on messages
10     push (LBTS_i(LBTS), server) in message buffer
11     send message_buffer to server
12     receive message from server
13    update FEL and LBTS
14   END IF
15 END WHILE
```

Algorithm 5-3: LEY: Server side

```
1  FOR each LP i
2      // buffer with messages destined for
3      // client i in current epoch
4      server_message_buffer[i] = []
5  END FOR
6
7  WHILE termination criteria not met
8    FOR each LP i
9     receive message_buffer
10     FOR message in message_buffer
11       update server_message_buffer[message.destination]
12     END FOR
13     update LBTS[i]
14   END FOR
15
16   compute LBTS = min_for all i (LBTS[i])
17
18   FOR each LP i
19     //piggyback LBTS on messages append LBTS in the
20     server_message_buffer[i]
21     send server_message_buffer[i] to LP i
22     server_message_buffer[i] = []
23   END FOR
24 END WHILE
```

5.4.3 Energy Consumption of LEY

We now analyze the energy consumed by LEY relative to the zero energy synchronization algorithm. With certain assumptions regarding energy consumption we show that LEY achieves zero energy synchronization for a certain class of distributed simulation applications.

Theorem 5-1: LEY in conjunction with the following assumptions yields the zero energy synchronization property:

a) *Each LP generates at least one remote event in each epoch.*

b) *Each LP receives at least one remote event in each epoch.*

c) *A constant increase in message size causes negligible increase in energy required for communication.*

Proof:. To prove that LEY is a zero energy synchronization scheme, it would suffice to show that LEY with the given assumptions consumes no more than *MinEnergy* energy to complete the application simulation. Hence the problem reduces to comparing the energy required by a distributed simulation program simulated with LEY to that of the *MinEnergy* value corresponding to the distributed simulation application.

We prove this by reducing LEY to an oracle-based implementation of the distributed simulation application, or simply O-DS. O-DS has the zero energy

synchronization property. Hence it consumes *MinEnergy* energy, with the assumption[7] that the oracle does not consume any energy.

We can prove the claim if we can construct LEY from an O-DS, such that following conditions are met.

1) The construction does not consume any energy, in other words any changes made to O-DS do not consume any additional energy.
2) The reduction relaxes the assumption that the oracle does not consume any extra energy.

Construction 1: Maintain a local variable, which stores the value last returned by oracle. Any available event is safe to process if it is smaller than this variable.

Construction 2: Constrain the sending and receiving of messages only when the LP blocks.

This constraint does not consume any additional energy because aggregation of messages as presented earlier can be assumed not to increase the energy consumed by the LP.

Construction 3: The oracle is consulted only when the LP blocks.

Construction 4: The oracle is implemented as in algorithm 5-4.

[7] Another implicit assumption here is that energy is not required for waiting. This follows from the definition of Minimum PDES energy.

Algorithm 5-4: LEY: Oracle

```
1 DEFINE oracle()
2   compute LBTS_i(LBTS)
3   append (LBTS_i(LBTS), server) in message buffer
4   send message_buffer to server
5   receive message from server
6   RETURN LBTS
7 END DEFINE
```

As noted earlier, from the definition of MinEnergy, synchronization messages are overhead whereas an event message is not. It must also be noted that, by *construction 2* and *assumptions* (*a)* and (*b)* the synchronization messages are always piggybacked on event messages. Hence by the implication of *assumption* (*c)*, synchronization messages do not consume additional energy. Thus reducing the effective energy consumption of the oracle to sending and receiving messages to and from server, which by *construction 2* forms the part of the simulation or in other words are not overhead.

Hence, we construct LEY with O-DS without consuming any more energy and relaxing the assumption of oracle with no extra energy. In other words, the constructions satisfy conditions 1 and 2. Hence LEY is zero energy given assumption a-c. This concludes the proof. ■

The main constraints needed to achieve zero energy synchronization are that the simulation application in each LP, or equivalently each processor, sends and receives at least one message in each epoch. For large simulations each processor will include many LPs. It is therefore reasonable to assume that there will be at least one message sent and received each epoch. The relation between τ and μ, as discussed in section 5.5.3, supports assumption c. Furthermore, as there will typically be many messages that are aggregated

together, this also implies that piggybacking timestamp information necessary to compute LBTS on such message exchanges would have negligible effect on the amount of energy consumed. In this sense, we view the constraints described above as mild constraints that will be applicable to many distributed simulation applications that arise in practice, especially large simulations with many LPs executing on each processor.

5.5 Implementation

Implementations of YAWNS, LEY, and an oracle-based synchronization mechanism were developed and used to evaluate the energy consumed by each approach for a sample application. Rather than implementing a general oracle mechanism applicable to any distributed simulation code, knowledge of the distributed simulation application was exploited in order to minimize the amount of energy required to complete oracle operations. This section describes these implementations as well as the sample application. The principal goals of this study were to determine:

a) The performance and energy consumed by the proposed synchronization scheme, LEY, relative to an implementation of YAWNS.

b) Assess the energy consumed by LEY relative to an oracle-based implementation approximating a zero energy synchronization algorithm.

We begin by describing the applications that are simulated to compare the synchronization schemes. Then follow this with a description of each of the implementations used for the study.

5.5.1 Applications

Two applications were used for this study. The first is the well-known Phold benchmark. The second is a simulation of a token-ring communications network. The applications were selected, in part, because highly efficient implementations of the oracle-based approach could be realized without resorting to the creation of a complete message log.

Phold is a widely used synthetic benchmark application by the distributed simulation community (Fujimoto 1989). We implement a Phold application with an infinite capacity server at each LP. The Phold application can be defined as follows. When processed, each event generates one new event, and the event so generated is sent to a randomly chosen remote LP at lookahead time in the future.

A ring network (also called as Token-ring network) is a unidirectional, Local Area Network. A ring network is formed by computing nodes connected to exactly two other nodes in the network with unidirectional links. Figure 5-1 shows a sample ring network with 4 nodes. Computing nodes on the network communicate with each other using tokens circulating in the network in one direction.

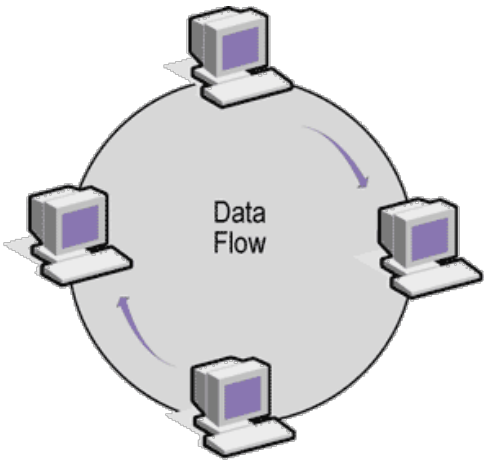

Figure 5-1: Token ring network topology.

Assume the nodes are identified as node 0 to node 3. For a node with ID i, the ID of the next node in the ring is given (i + 1) mod 4. Although each node can directly send messages to only one node in ring network, a token is used to send messages to other nodes. Suppose node 0 sends a message to node 2. To send data to a specific node, the sender must first wait for the token to arrive. When received, the token is checked if it is free. If it is free the sender marks the token as being used and writes the data for the message along with the destination ID and its own ID as the source ID into the token. The sender then forwards the token to next node in the ring. The token is then passed on from one node to next in the ring, until it arrives at the destination node, in this case node 2. The recipient node reads the data and marks the token as an acknowledgement, swaps the sender and destination ID, and sends the modified token to the next node in the ring. Eventually the token is received by the sender, which sees the acknowledged token and modifies its status to indicate it is a free token and transmits the token back into the ring.

5.5.1.1 Implementation Details for Ring Network

We simulate a multi-token ring network, similar to one presented in (Kamal 1990), with each LP simulating a node in the network. We make some assumptions to simulate the ring network. First, we assume that the nodes can process only one token at a time. This is consistent with our earlier note concerning single unit-capacity server LPs. This implies that if multiple tokens arrive simultaneously, they would be processed sequentially, in contrast to a less pragmatic infinite capacity LP. Second, the nodes take a constant amount of time to processes a token. This assumption can be rationalized by the fact that the frequency of network hardware is much slower than that of the processors. Third, we assume uniform links and nodes. Fourth, we assume constant size tokens, consistent with previous assumption of a constant propagation delay for all links.

Suppose that at time t a token is processed in an LP. A remote event is generated for the LP simulating the next node in the ring, with a timestamp of $t + processing\_time + transmission\_time$ and the simulation time of the LP advances to $t + processing\_time$. If the token arrives while the node is busy, it is queued and is processed sequentially in time stamp order.

Each node is initialized with a constant number of tokens at simulation time $t = 0$. Given the application, the total number of tokens in circulation remains constant throughout the execution of the simulation. The LPs terminate if they reach the pre-specified maximum simulation time or the simulation as a whole has exhausted all the events that can be processed before the maximum simulation time.

Table 5-1: Simulation Application Constants

Name	Value
Maximum simulation time (T)	10^6 units
Token processing time (p)	1 unit
Transmission time (t)	10 units
Number of initial tokens per node (I)	10

The empirical evaluation focused on the effect of varying the number of nodes in the simulation or more specifically a weak scaling of the application. Table 5-1 summarizes the constants used for the study.

5.5.1.2 Analysis of Ring Network Application

Given the multi-token ring network application, described in previous section, total number of events processed by a LP, E, for a given max simulation time, T, is given by:

$$E = \begin{cases} \left\lceil \mathrm{T\ x}\ \dfrac{I}{t+p} \right\rceil, & \text{if } \mathrm{t + p > I\ x\ p} \\ \left\lceil \mathrm{T\ x}\ \dfrac{1}{p} \right\rceil, & \text{if } \mathrm{t + p \leq I\ x\ p} \end{cases} \tag{4}$$

Where, I is the number of initial tokens per node, p is the token processing time and t is the transmission time. With the constants as mentioned in table 1, *E* should be *909091*. All the implementations were verified against this value of E.

5.5.2 LEY and YAWNS Implementations

Implementations of LEY and YAWNS were created for this study. These are described next.

5.5.2.1 LEY

We use the client server based LEY implementation presented earlier in Algorithms 5-2 and 5-3. For the specific application of the ring network simulation and the assumptions as stated in the previous section, further optimizations were considered. For example, the destination of any generated event is always the ID of the next node in the ring. Hence the destination tag was removed from the message, as the server can identify the destination based on the sender ID.

5.5.2.2 YAWNS

To be consistent across all variations of the synchronization protocols being studied, algorithm 5-1 was adapted in a client server implementation of YAWNS. Algorithm 5-5 and 5-6 present the adapted version of YAWNS for the ring network simulation. Each LP in the simulation is a client, which executes algorithm 5-5. As is common in client server infrastructures clients communicate with each other through the server. Again, application specific optimizations, such as skipping the destination field in the event message, were implemented.

It should be noted here that although the synchronization part of the implementation is generic and can apply to any distributed simulation application, the communication pattern is specific to the applications described earlier.

Algorithm 5-5: YAWNS: Client side

```
1  LBTS = 0
2  WHILE termination criteria not met
3    IF(FEL not empty && timestamp of FEL.top <= LBTS) THEN
4      process and remove top event
5      send generated event to server as event_message
6      receive any waiting events from server
7      push the received events in FEL
8    ELSE
9      compute LBTS_i(LBTS)
10     send LBTS_i(LBTS) as a synchronization_message
11     receive LBTS from server
12     update LBTS
13     receive any waiting events from server
14     push the received events in FEL
15   END IF
16 END WHILE
```

Algorithm 5-6: YAWNS: Server side

```
1  Initialize server_message_buffer for each LP
2  LBTS = ∞
3  WHILE termination criteria not met
4    LBTS[i] = ∞, FOR each LP i
5    receive message from any client (say, LP i)
6
7    IF event_message THEN
8      push the message to the buffer for destination LP
9      send any message in server_message_buffer[i] to LP i
10     free server_message_buffer[i]
11   END IF
12
13   IF synchronization_message THEN
14     update LBTS
15     IF  all clients have sent synchronization messages
16         for this epoch THEN
17       send LBTS values to all clients
18       reset LBTS to infinity
19     END IF
29   END IF
30 END WHILE
```

5.5.2.3 OLEY: Synchronization with Oracle

In terms of LEY, whenever an LP blocks, i.e., needs to synchronize, an oracle is consulted to determine the LBTS value for the next epoch. In relation to the definition of the oracle presented earlier (see equation 1), consider the definition of the LBTS presented in section 5.4.1 and equation 3. LBTS provides the lower bound for any event that can be created by any LP in the next epoch. Hence LBTS is also a lower bound for each individual LP. Therefore, the LBTS value at any time can be treated as a valid oracle output.

There are several approaches to implementing the oracle. A general approach is to log every message sent in the simulation in a pre-simulation run and then refer to this log while simulating. However, there is an inherent, potentially significant energy cost of this oracle related to the maintenance and access of the log. A more efficient way to implement oracle is to exploit any application properties to determine the LBTS value for each epoch.

For our Phold application, the LBTS value for each epoch increments exactly by the lookahead amount. This results from the infinite capacity of the server.

For our ring network implementation, we use the maximum of timestamps of the event received for an epoch as the LBTS value for the epoch. As can be inferred, this value holds due to the communication pattern and the symmetric nature of the application.

As was pointed out earlier in section 3, the setup of the baseline implementation (in this case synchronization with the oracle) is important. Hence rather than YAWNS, we modify the LEY algorithm to yield OLEY, the implementation of LEY using the oracle. In terms of implementation, an OLEY client would replace the $LBTS_i$ (LBTS) computation and piggybacking steps in LEY client (Algorithm 5-2) with a request to oracle. Similarly, an OLEY server would skip the steps to compute and piggyback LBTS in LEY server (Algorithm 5-3).

5.5.3 Analysis of LEY, YAWNS and OLEY

To make the discussion of the performance analysis applicable for a generic communication network among the LPs, let us consider the following communication model. The energy costs associated with communication of the messages are considered to be composed of two components. First component is the energy cost associated with initialization of a message communication. This consists of the energy required for the setup of the message as it makes its way through different application and network layers all the way down to the physical layer. Any energy costs incurred due to changes in the state of the hardware are also attributed to the initialization. Initialization cost is assumed to be independent of the length of the message itself. The underlying assumption is that the messages are smaller than maximum transmission unit (MTU) of the underlying communication channel and protocol. This assumption generally holds because the MTU for Token Ring is 4464 Bytes, for IEEE 802.11 Wi-Fi is 2304 bytes and that of Ethernet is 1500 - 9000 bytes. Let this cost be denoted by τ. The second component of the energy cost is the energy required for transmitting a bit of the message. Let this cost be denoted by μ. The size of the headers added to any message is assumed to be constant and the

energy required for the transmission of the header bits is considered to be included in the initialization cost τ. Depending on the underlying communication model, τ and μ will include the costs related to retransmissions and packet loss. Hence, a message of size m bits would incur an energy cost of τ + mμ. Although the exact empirical values would depend on the underlying software and hardware architecture, the initialization cost τ is about 3 to 4 orders of magnitude larger than the transmission cost μ. For instance, for the platform of experimentation used in this chapter, τ was found to be of the order 10^{-4} and μ to be of the order 10^{-8}. Hence for analysis purposes we assume τ >> μ. To simplify the following discussion, for the following analysis it is assumed that reception of messages does not consume energy. This assumption will be relaxed after the analysis.

In terms of the properties of the synchronization algorithm and simulation application, let's assume that LEY completes a simulation in K epochs. It is easy to see that YAWNS and OLEY would also require K epochs to complete the simulation. Let $\grave{n}_{i,k}$ and $n_{i,k}$ denote the number of events processed and the number of events generated by LP i in k^{th} epoch, respectively[8]. Similarly, let total number of events generated by an LP be N_i, where $N_i = \sum_{k=0}^{K-1} n_{i,k}$ and total number of events processed be $\grave{N}_i$. Also, let each individual synchronization message be of size m and event message be of size m'. Next, let $p_{i,j}$ be the probability that an event generated by LP i is scheduled in LP j. From this definition, the probability of remote event (PoRE) for LP i is $p_i = 1 - p_{i,i}$. In addition, assume the energy cost of computations related to synchronization is constant; let it be denoted by α. As the simulation application and platform is same across all the

[8] It can be noted that depending on the simulation, it might be possible to describe $n_{i,k}$ and $\grave{n}_{i,k}$ in terms of each other. We retain the general form for our discussions.

cases, the energy cost of computation of event $C_{i,j}$ is equal for all implementations. Let the energy required for computing an event be β. Finally, as all the implementations and algorithms discussed in the previous sections are client server based, hence $S_{i,j}$ is non zero for all events.

The YAWNS algorithm exchanges event messages individually and sends an extra message for synchronization. Hence for a simulation synchronized by YAWNS, the energy required for k^{th} epoch by i^{th} LP is

$$E_{YAWNS}^{i,k} = \text{MinEnergy}^{\text{i,k}} + synchronizationEnergy_{YAWNS}^{i,k}$$

$$= (\beta \text{n}_{\text{i,k}}^{`} + \tau p_i n_{i,k} + \mu p_i n_{i,k} m') + (\tau + \mu m + \alpha)$$

$$= \beta \text{n}_{\text{i,k}}^{`} + \alpha + \tau(p_i n_{i,k} + 1) + \mu\left(m' p_i n_{i,k} + \text{m}\right)$$

Summing over all the epochs gives us the energy required by i^{th} LP

$$E_{YAWNS}^{i} = \sum_{k=0}^{K-1} E_{YAWNS}^{i,k}$$

$$= \sum_{k=0}^{K-1} (\tau(p_i n_{i,k} + 1) + \mu\left(m' p_i n_{i,k} + \text{m}\right) + \beta n_{i,k}^{`} + \alpha)$$

$$= \tau\left(p_i \times \sum_{k=0}^{K-1} n_{i,k} + K\right) + \mu\left(\text{m}' p_i \times \sum_{k=0}^{K-1} n_{i,k} + \text{Km}\right) + N_i^{`}\beta + K\alpha$$

$$= \tau(p_i N_i + K) + \mu(\text{m}' p_i N_i + \text{Km}) + N_i^{`}\beta + K\alpha \quad (5)$$

On similar lines, the energy required by i[th] LP in a simulation synchronized by LEY is as follows. For LEY all the event message communication occurs at the end of the epoch and the synchronization message is piggybacked over the event messages.

$$E_{LEY}^{i,k} = MinEnergy^{i,k} + synchronizationEnergy_{LEY}^{i,k}$$

$$= \beta n_{i,k}^{`} + \tau + \mu\left(m' p_i n_{i,k} + \mathrm{m}\right) + \alpha$$

$$E_{LEY}^{i} = K\tau + \mu(\mathrm{m}' p_i N_i + \mathrm{Km}) + N_i^{`}\beta + K\alpha \quad (6)$$

Finally, the energy required by LP of simulation synchronized by OLEY is

$$E_{OLEY}^{i} = K\tau + p_i N_i m'\mu + N_i^{`}\beta + K\alpha \quad (7)$$

To relax the assumption that receiving messages do not consuming energy, we introduce one more property of the application. Let r_j be the probability that a remote event generated in the simulation is for LP j. Assuming there are *L* LPs in the simulation, it can be derived that $r_j = \frac{\sum_{i \neq j, i=0}^{L-1} \frac{p_{i,j}}{p_i}}{L}$. Finally, assuming the finalization cost (analogous to initialization cost) and the reception cost (analogous to transmission cost) are respectively similar to τ and μ. Then Equations 5, 6 and 7 can be rewritten as,

$$E_{YAWNS}^{i} = (r_i N^r + p_i N_i + 2K)\tau + ((r_i N^r + p_i N_i)m' + 2Km)\mu + N_i^{`}\beta + K\alpha \quad (8)$$

$$E^{i}_{LEY} = 2K\tau + ((r_i N^r + p_i N_i)m' + 2Km)\mu + N_i^{`}\beta + K\alpha \quad (9)$$

$$E^{i}_{OLEY} = 2K\tau + (r_i N^r + p_i N_i)m'\mu + N_i^{`}\beta + K\alpha \quad (10)$$

Here N^r is the total number of remote events generated during the simulations, i.e. $N^r = \sum_{i=0}^{L-1} p_i N_i$.

5.6 System Configurations

All simulations were developed in C++ with all communications using MPI. As was mentioned earlier, all simulations were developed with a client server architecture, where each LP was mapped to a client. In terms of implementation, each client and the server were mapped to individual MPI processes. MPI processes were assigned in a round robin fashion among the available CPU cores.

The experiments were performed on a micro-cluster platform designed for mobile, high performance computing. The micro-cluster is comprised of NVIDIA's Jetson TK1 development boards. Each development board consists of a Tegra TK1 SOC including NVIDIA's 4-Plus-1™ Quad-Core ARM® Cortex™-A15 32-bit CPU with 4 cores operating at 2.3 GHz and 2 GB memory. Each of these boards runs Ubuntu 14.04.5 LTS. The boards communicated over an Ethernet LAN. Two such development boards were used for the study. Energy and power measurements were performed using a PowerMon2 power measurement system (Bedard, Fowler et al. 2009).

Power and energy values were measured for one of the two boards, called the board of interest. The server was always assigned to a process in a different development

board from the board of interest. This ensures that energy values include only LPs. This is consistent with an implementation where simulations reside on edge nodes e.g. smartphones. In this client-server architecture all LPs assigned to the board of interest communicate exclusively using inter-board communications. The setup and the round robin LP assignment result in two experimental scenarios. In the first scenario, the number of LPs is even so the number assigned to each board is the same. In the second scenario the number of LPS is odd and the board of interest has one more LP than the other board.

5.7 Results

In this section we present the results of the empirical study using the implementations of the applications discussed earlier. The experiments were designed to provide insight into the energy consumption behavior of LEY, YAWNS and OLEY. The applications were weakly scaled to study the effect of increase in number of LPs on the metrics of interest.

5.7.1 Principle Metrics

For these experiments YAWNS, LEY, and OLEY were compared using two principal metrics, energy consumption and execution time:

1. **Energy consumption:** The instantaneous power consumed by one of the development boards (called, the board of interest) was aggregated over the duration of the simulation execution and used to determine the average power

consumption. This value is multiplied by the execution time to determine the total amount of energy consumed.

2. **Execution time:** Execution time indicates the time required for the application to complete its execution. This metric was reported using the C++ class, std::chrono::high_resolution_clock.

We restrict our experiments to a maximum of 7 LPs (8 processes). This is due to the limited number of available cores on the platform used for these experiments.

Before moving on to comparison and discussion of energy consumption, figures 5-2 and 5-3 present the average power consumed and the time required for ring network application as the number of LPs in the system increases.

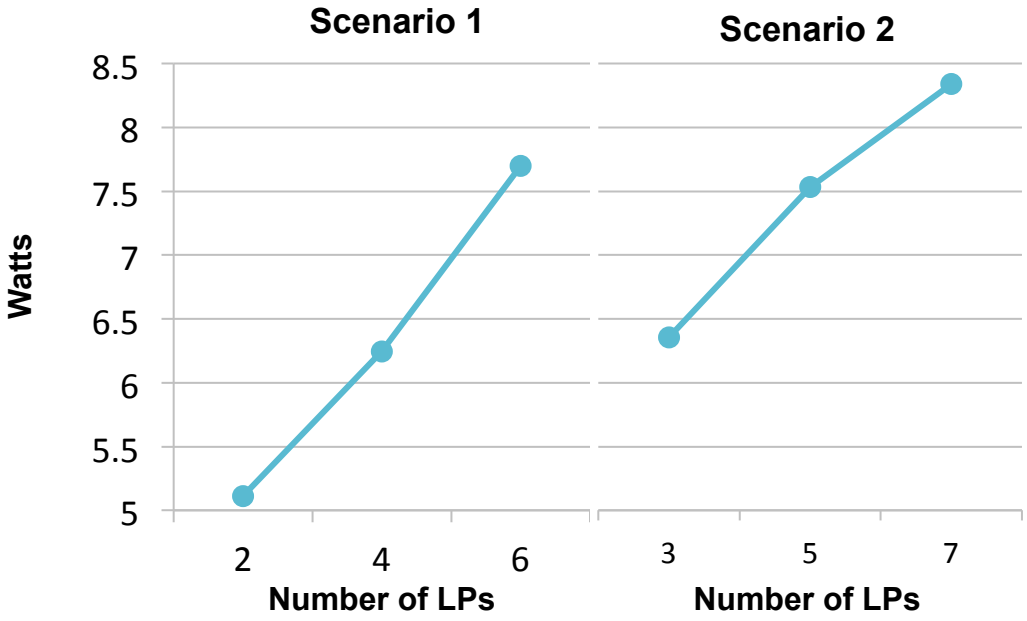

Figure 5-2: Average power consumed by the ring network application synchronized with LEY.

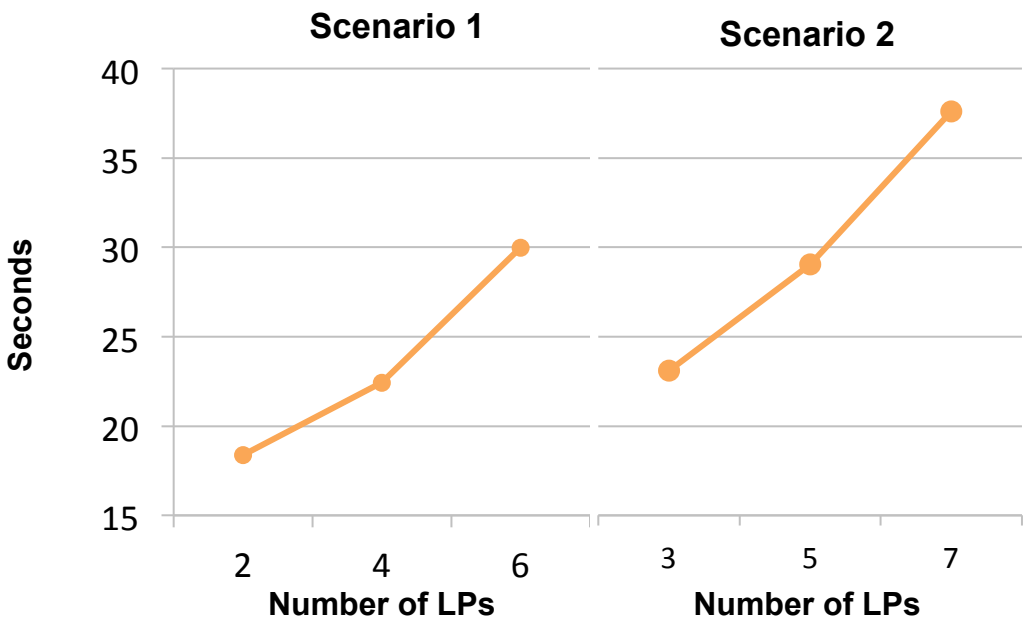

Figure 5-3: Execution time for the ring network application synchronized with LEY.

Figure 5-4 shows the results from energy consumption measurements for the Phold application. The amount of energy consumed largely increases with the number of LPs as one might expect, as noted below. More importantly, these measurements indicate that LEY consumes much less energy than the unoptimized version of YAWNS. This is also corroborated by energy consumption measurements of the ring network application. Further, it can be seen that LEY's energy performance approaches that of the oracle-based scheme suggesting energy performance approaching optimal.

Figure 5-5 shows the energy consumption of the ring network simulation. Again, LEY consumes much less energy than the original YAWNS implementation, and approaches that of the oracle-based implementation. Here we see the energy consumption does not increase uniformly as the number of LPs increases. This behavior is seen to a lesser degree in the Phold measurements. Upon closer inspection one observes that the trend of increasing energy occurs when only an odd or even number of LPs is considered

separately. This difference in the odd and even number of LPs can be bolstered by the round robin assignment of LPs to processes (also, cores). With two boards, an even number of LPs are distributed evenly among the boards, but an odd number of LPs causes an imbalance. This in conjunction with the uniform nature of the application leads to the observed trend.

An important observation, common to both the applications, is that the optimized synchronization scheme leads to a reduction in execution time proportional to the reduction in energy. This follows from the fact that the difference in average power consumption of the three synchronization schemes is relatively small. Another important observation is that for both applications, LEY achieves energy consumption that is only slightly greater than that of OLEY, suggesting that practical realization of zero energy synchronization of distributed simulation codes may be feasible.

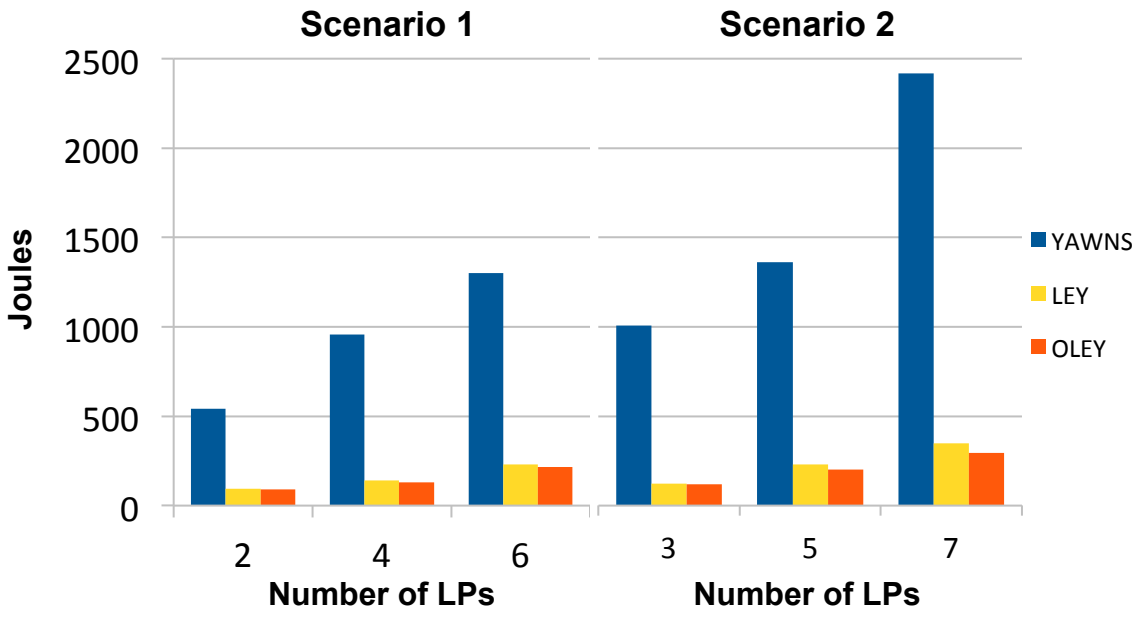

Figure 5-4: Energy consumed by the Phold application.

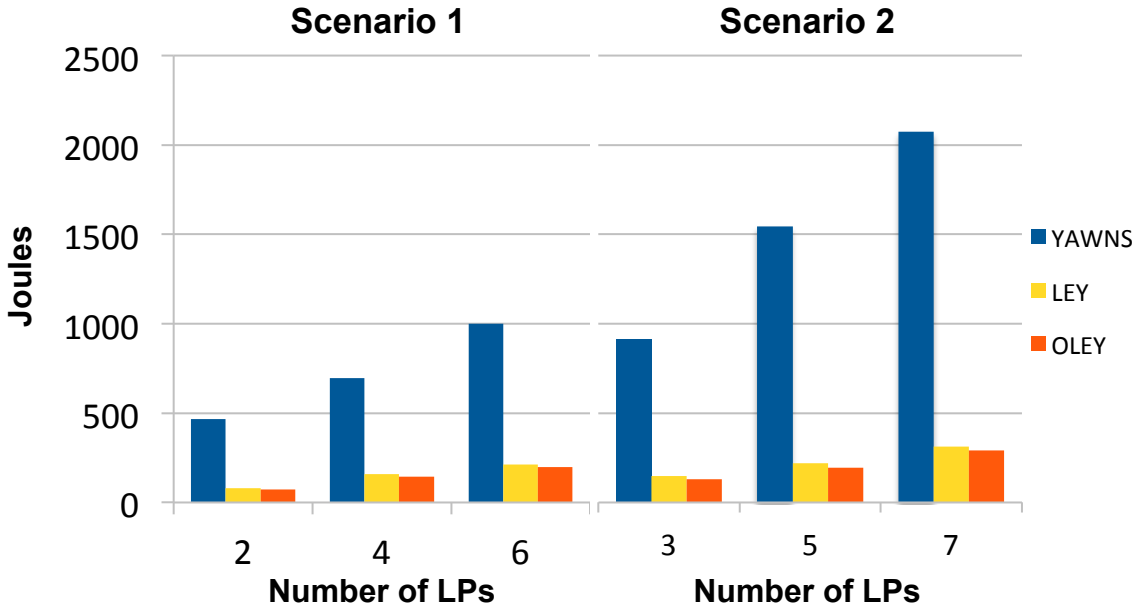

Figure 5-5: Energy consumed by the ring network application.

5.7.2 Second Order Metrics

The large difference in the energy consumed by YAWNS and LEY highlights the fact that the optimizations introduced in LEY provide both execution time and energy consumption benefits, relative to an energy-oblivious implementation of YAWNS. However, these differences are difficult to quantify.

We characterize the performance advantage in terms of energy as *energy improvement*. *Energy improvement* is defined as the percent decrease in the amount of energy consumed by LEY when compared to that of YAWNS. Similarly, we define the synchronization overhead of a synchronization algorithm with respect to energy as per cent increase in energy required by a simulation when compared to that of the simulation with an oracle.

More precisely, the energy overhead for LEY and improvement of LEY over YAWNS is given by equations 5 and 6.

$$Energy\ Overhead = \frac{E_{LEY} - E_{oracle}}{E_{oracle}} \times 100 \tag{11}$$

$$Energy\ Improvement = \frac{E_{YAWNS} - E_{LEY}}{E_{YAWNS}} \times 100 \tag{12}$$

Where, E_{LEY} and E_{YAWNS} is the energy required for simulating the application with LEY and YAWNS, respectively. E_{oracle} is the energy required for simulating the application with an oracle, or in this case OLEY.

5.7.2.1 Expected Energy Overhead and Energy Improvement

Substituting equation 9 and 10 in 11, we get

$$Energy\ Overhead = \frac{2Km\mu}{2K\tau + (r_i N^r + p_i N_i)m'\mu + \grave{N_i}\beta + K\alpha} \times 100 \tag{13}$$

For our application $m = m' = 32$ bits. As μ << τ, we have μm ≪ τ. Hence energy overhead is expected to be very small. It can be noted that this is independent of the number of epochs, energy required for computations, PoRE value and number of events.

Substituting equation 8 and 9 in 12, we get

$$Energy\ Improvement = \frac{(r_i N^r + p_i N_i)\tau}{(r_i N^r + p_i N_i + 2K)\tau + ((r_i N^r + p_i N_i)m' + 2Km)\mu + \grave{N_i}\beta + K\alpha} \times 100 \tag{14}$$

It can be noted here that if PoRE value, p_i, is zero then energy improvement is zero. Furthermore, as noted above $m = m' = 32$ bits, hence μm ≪ τ, $\tau(pN_i + K) \gg$

$\mu m(pN_i + K)$. Also, for our applications, $p_i = 1$, $N_i = N_i^{`}$ and $r_i N^r = N_i$. Also, for our platform of experimentation, $\tau \gg \beta$ and $\tau \gg \alpha$, we have $N_i\tau \gg N_i^{`}\beta$ and $K\tau \gg K\alpha$. Hence,

$$Energy\ Improvement \approx \frac{N_i}{N_i + K} \times 100 \tag{15}$$

As K depends on the value of lookahead, so would Energy Improvement. In case of the ring network application with everything else constant, K is inversely proportional to lookahead, hence it is expected that the Energy Improvement would increase with lookahead. Substituting ring network application specific empirical values, we get,

$$Energy\ Improvement \approx \frac{909091}{909091 + 90910} \times 100 = 90.9$$

5.7.2.2 Empirical Energy Overhead

Figures 5-6 and 5-7 show the *energy overhead* of LEY, as defined above, with respect to OLEY. It can be seen that the overhead is modest. For the Phold application the increase in energy with the increase in the number of LPs is due in part to the random event distribution. In particular, assumptions (*a)* and (*b)* of theorem 1 does not hold for this application. As there might be epochs in which LPs do not receive or send any event. This causes extra message communications for synchronization which otherwise would not have been required when compared to the *MinEnergy* energy needed by the oracle based simulation.

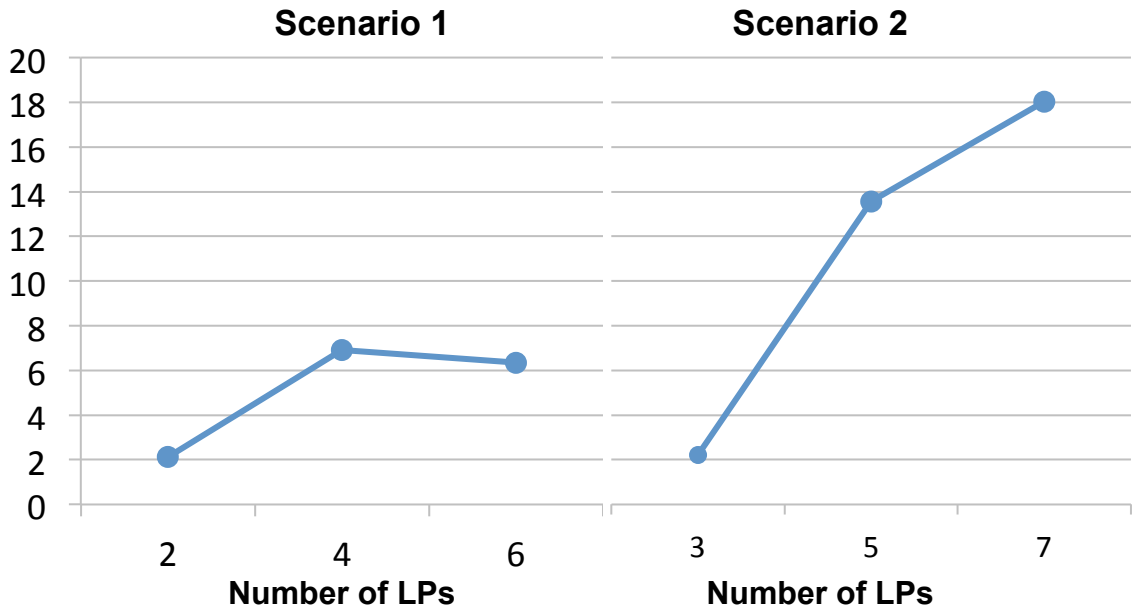

Figure 5-6: Overhead of LEY for the Phold application increases as the number of LPs increase.

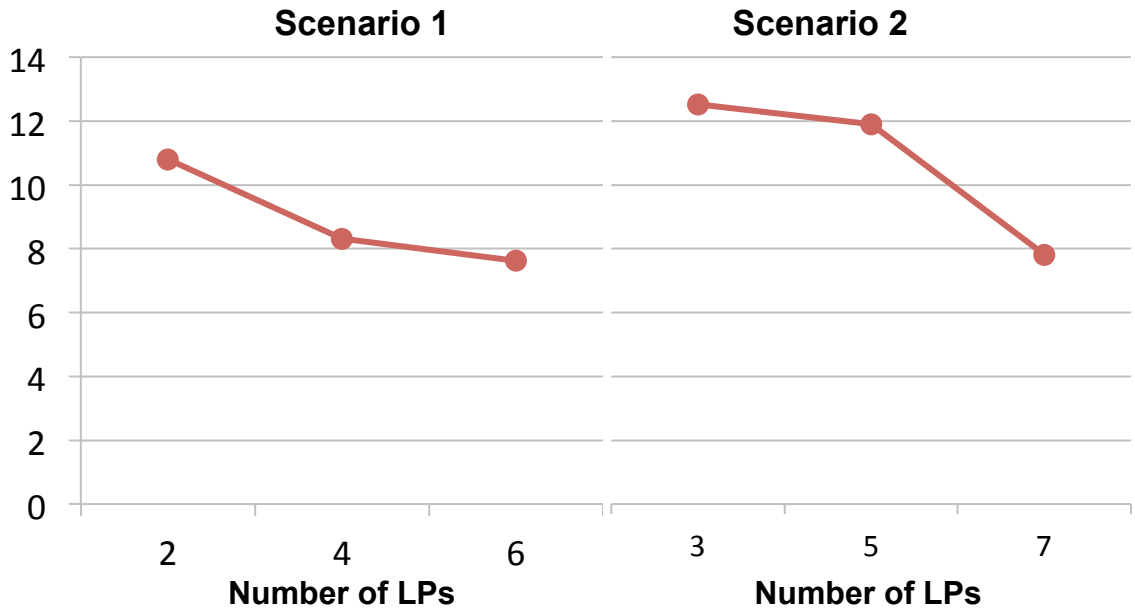

Figure 5-7: Overhead of LEY for the Ring network application is minimal and the overhead decreases as the number of LPs increase.

For the ring network application, the LPs have messages to send and receive in each epoch resulting in the LEY measurements to be closer to the energy consumption of OLEY. The small energy overhead is a result due to the real-world implications of assumptions such as the processor does not consume energy in idle state. The interesting irregular nature of the plots can be explained as a culmination of the idle energy and the imbalance due to the round robin distribution of the LPs.

The main conclusion of the results presented for this metric is that the energy overhead of LEY is very close to that of OLEY and that the deviation from the assumptions of theorem 1 may cause a minor increase in the overhead.

5.7.2.3 Empirical Energy Improvement

Figures 5-8 and 5-9 presents the energy improvement of LEY. We see about an 84% energy improvement for Phold and an average of 82% energy improvement for the ring network. In both cases the energy improvement is modestly affected by the increase in the number of LPs. The nature of energy improvements using LEY with the ring network can again be attributed to the assignment of LPs. The increase in the energy consumption of YAWNS is in proportion with that of LEY for an even number of LPs but the increase is relatively higher for odd number of LPs, as the board of interest in the latter case has more LPs than the other board.

The conclusion for this metric is that LEY outperforms YAWNS by a large magnitude, which outweighs the possible small increase in the overhead of LEY. Also, the improvement can have minor dependence on the application being simulated.

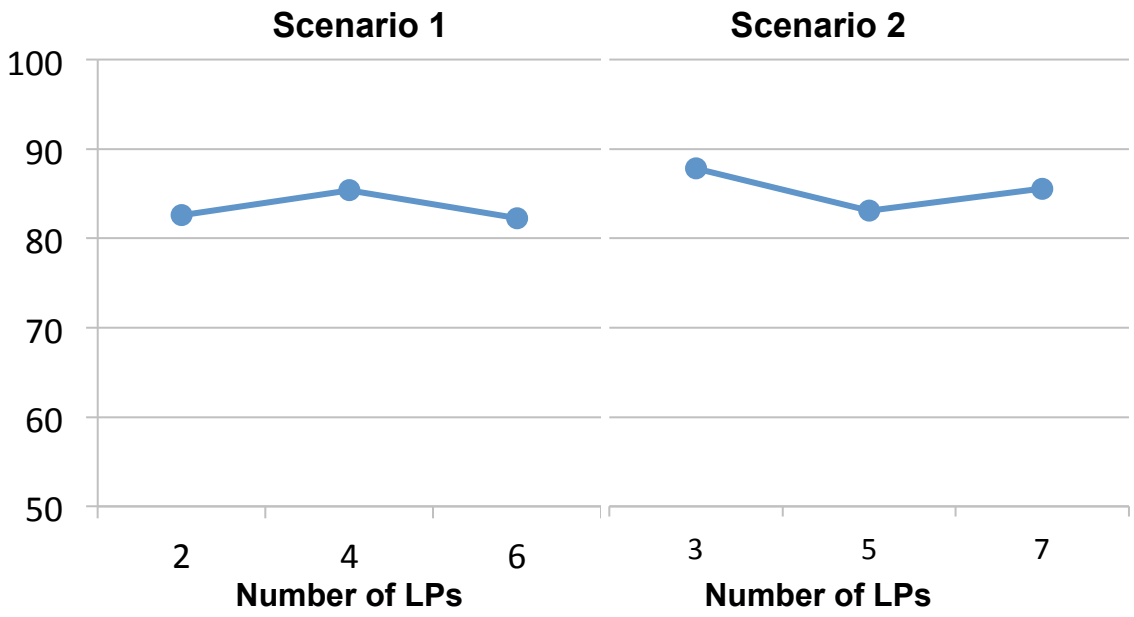

Figure 5-8: LEY has an average 84% energy improvement for Phold.

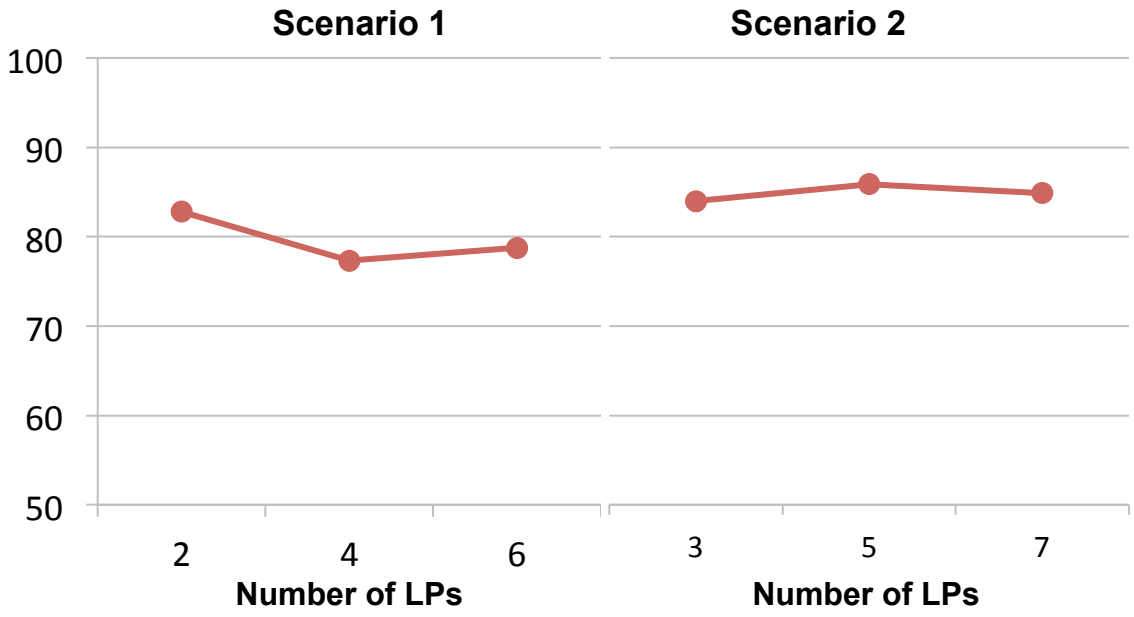

Figure 5-9: Ring network with LEY has an average energy improvement of 82%.

5.8 Conclusions

In this chapter we highlight the importance of energy consumption in distributed simulation as an important challenge facing the community. Observing that synchronization does not directly contribute to computational results produced by the simulation, we propose zero energy synchronization as a goal that distributed simulation algorithms and implementations might strive to achieve. We propose the use of an oracle both as a theoretical construct that can be used to analyze the energy required for synchronization as well as a practical method that can be used to measure energy costs associated with synchronization. To explore the feasibility of achieving zero energy synchronization, Low Energy YAWNS (LEY) is proposed. We prove that with some mild assumptions LEY can achieve zero energy synchronization for many distributed simulation codes.

To provide tangible evidence of the practicality of this work, an experimental study was completed. Empirical measurements of LEY, an implementation of YAWNS not optimized to minimize energy consumption, and an oracle based implementation designed to measure the amount of energy required to complete a simulation with a zero energy synchronization algorithm with two applications we found that LEY improves the energy consumption of the un-optimized YAWNS implementation by about 84% and incurs only a small, approximately 10% additional energy cost compared to an oracle-based approach. Execution time results showed similar improvements.

The experimental results presented here suggest that zero energy synchronization is a reasonable goal in realizing energy efficient distributed simulation codes that may be

achievable in practice. We caution, however, that these results correspond to only preliminary experimentation with the synchronization algorithm, and a more thorough and comprehensive analysis and experimental study is required to draw definitive conclusions.

The zero energy synchronization concepts discussed in this work, along with the methodology utilized, suggest an approach to measure and evaluate the energy consumption of distributed simulation synchronization algorithms. We believe this approach could be useful for further research in the development of energy efficient parallel and distributed simulations.

The assumptions for zero energy synchronization point towards avenues for further improvement of the synchronization schemes to approach zero energy synchronization in real life application scenarios. For example, the assumption that energy is expended in only active state might not be possible in real-world systems due to say, switching overheads, but the schemes that can optimize these pockets of energy use can further reduce the overhead.

6 QUEUEING NETWORK SIMULATION USING COMPOSITE PARALLEL PREFIX

In this chapter we explore another approach of improving the performance of parallel and distributed simulations. More specifically, the impact of using non-conventional and specialized simulation platform on the performance of a parallel simulation is explored. For this purpose, a data parallel algorithm for parallel and distributed simulation of queueing network is proposed. The proposed algorithm is used to conduct parallel simulation of a queueing network conventional (CPU based) platform and non-conventional (GPU based) platform. Furthermore, composition of parallel prefix is discussed to further improve the energy efficiency of the proposed algorithm, emphasizing the significance of efficient algorithms.

Queues have long been used to model and understand the behavior of many real-world and synthetic systems. Examples include vehicle traffic analysis (Huang, Hunter et al. 2010) computer systems performance, and computer network modeling (Chiu, Dumont et al. 1975, Bard 1977, Luo and Lu 2000). The author of (Kleinrock 1975) describes queueing system as any system where arrivals place demands on a finite capacity resource, highlighting the wide applicability of queueing networks for analysis and system optimization. Queueing network simulations, especially for large systems can be expensive computationally.

With the growth and increased availability of the distributed and parallel computing platforms, the parallelization of queueing network simulations is of great interest. Parallel simulations of queues are often conducted using discrete event

simulation by distributing the simulation in sub-components (logical processes). Although this approach is applicable to a large gamut of simulations and analyses, it is limited in terms of the amount of parallelism that can be exploited. In terms of the queueing systems, the degree of parallelism is often limited to the number of queues. This work discusses a recurrence-based approach. Such an approach increases the degree of parallelism by using parallel prefix algorithms and variations of the same.

Parallel prefix scan operations can be found at the heart of application a plethora of applications. The prefix scan problem, as described in (Blelloch 1990), can be stated as the problem of computing of the sequence $X = \{x_0, x_1, \dots, x_n\}$ given another sequence $A = \{a_0, a_1, \dots, a_n\}$ such that

$$x_i = \begin{cases} a_0 & i = 0 \\ x_{i-1} \oplus a_i & 0 < i < n \end{cases} \quad (1)$$

Where $\oplus$ is any binary associative operator. Various versions of this recurrence relation have been put forth to solve a multitude of problems in various domains of science and engineering. Some examples of applications could be buffer allocation, radix sort, quick sort, data compression, N-body simulation, sequence alignment and many more.

In this chapter we propose a data parallel algorithm for parallel and distributed simulation of queueing network and describe composition of parallel prefix operation. Composition of parallel prefixes can be used to optimize a series of parallel prefix computations, where the output of one parallel prefix computations forms the input for the next in the series[9]. The composition is described for the series with the property that the operators for parallel prefix distribute over the operators of the subsequent parallel

[9] If the subsequent parallel prefix in the series is independent from the current, then the series can also be trivially composed with the described approach.

prefixes in the series. Then we derive the data parallel algorithm for parallel and distributed simulation of queueing network, more specifically queuing networks composed of FCFS G/G/1 queues. The performance advantages of parallel queueing network simulation and that of composition in terms of execution time and energy consumption are demonstrated using empirical studies.

The chapter is organized in a similar order as the description presented in the last paragraph. In the following section we discuss prior research related to the major concepts related to this chapter. Then we briefly describe the parallel prefix (or scan) and build up to the description and analysis of composite parallel prefix. Then we move on to the queueing network simulation and discuss an algorithm for computing the departure time of jobs for parallel and distributed simulation of queues. This discussion is then followed by the description of the goals and setup of the empirical studies. Next, we discuss the results of the empirical study and finally close with a discussion of results, a special case of queues and conclusion.

6.1 Related Work

6.1.1 Parallelizing Queueing Network Simulations

Approaches to parallelize queueing network simulations can be broadly categorized in two major categories. The first approach views queueing networks as a special case of discrete event simulation applications. This approach has been presented in works such as (Lin and Lazowska 1991, Huang, Hunter et al. 2010, Park and Fishwick 2010, Huang, Fujimoto et al. 2013). These approaches, which range from ones that use specific hardware to general hardware platforms, have the advantage of being applicable

to larger set of application and scenarios. But as mentioned earlier, they cannot fully utilize the available parallelism. The second type of approach is specific to queueing networks and is the focus of application study of this chapter, for example (Greenberg, Lubachevsky et al. 1990, Lin and Lazowska 1991, Heidelberger and Nicol 1993). Studies such as these are specific to queues, and sometimes specific classes of queues. Work presented in (Wang and Abrams 1992) looks at time parallel approach for FCFS queues with losses, that is the jobs are dropped if queue is full. Work described in (Heidelberger and Nicol 1993) accelerates a specific class of queues arising in communication and distributed computing systems using conservative uniformization.

6.1.2 Parallel Prefix Scan

Due to the importance and wide applicability of the parallel prefix computations it has received attention both in terms of algorithmic and implementation specific development. One of the earlier for the parallel implementation of prefix scan appears in (Hillis and Steele Jr 1986) (Blelloch 1989). Later works proposed other optimizations of the parallel prefix algorithms. A multitude of works looks at the application of parallel prefix. Author in his work (Blelloch 1990) discuses a series of modification and applies parallel prefix to applications like quick sort. In (Ladner and Fischer 1980), the authors use prefix scans to efficiently simulate a finite state transducer. More recently optimization for parallel prefix have been proposed for their implementation on parallel and distributed environments (Sanders P. and J.L. 2006, Sengupta, Harris et al. 2007, Sengupta, Harris et al. 2011, Yan, Long et al. 2013). Library implementations of the parallel prefix scan exists in major parallel computing packages like MPI, NVIDIA's CUB and Thrust.

6.1.3 Power and Energy Efficiency

Energy and power, as mentioned earlier has become a very important constraint. There is a substantial literature in power- and energy-aware computing systems, and a variety of techniques may be employed. Dynamic voltage and frequency scaling (DVFS) and power capping and the combination thereof have resulted in a range of studies to reduce power and energy consumption (Hua and Qu 2003, Ge, Feng et al. 2005, Freeh, Lowenthal et al. 2007). Another area of focus is improving work efficiency by scheduling e.g. Charm++ (Acun, Langer et al. 2016) and (Garzón, Moreno et al. 2017). Finally, there are application specific studies such as (Karamati, Young et al. 2018) and those for parallel and distributed simulations which either look at the simulation application (Neal, Fujimoto et al. 2016) or the underlying infrastructure like synchronization mechanism (Child and Wilsey 2012, Fujimoto and Biswas 2015, Biswas and Fujimoto 2016, Fujimoto, Hunter et al. 2017, Biswas and Fujimoto 2018).

6.2 Composite Parallel Prefix

Parallel implementation of prefix scan algorithm, or simply parallel prefix algorithms, are algorithms that can be used to compute the prefixes over distributed or shared memory processes in parallel. Although a sequential algorithm can be easily developed and implemented, the parallel version of prefix sum problem is not trivial due to the dependence on previous elements. The sequential prefix scan can be completed in $O(n)$ time. Many parallel prefix algorithms have been proposed, for simplification of discussion in this section we use the one proposed by Hillis and Steele (Hillis and Steele

Jr 1986). It must be noted that all the discussions can be easily carried forward to other implementations like Blelloch's (Blelloch 1990) implementation. Algorithm 6-1 lays out the parallel prefix algorithm for the case where n processes are used to compute the recursion presented in equation 1. As is common, a simplifying assumption here is that n is an integral power of two. Also, line 4 of algorithm 6-1 for a distributed memory parallel system would result in a message exchange from process $\left[\mathrm{k} - 2^{j-1}\right]$ to process k.

Algorithm 6-1: scan(x,⊕)

```
1 FOR j = 1 to log2 n DO
2  FOR all k in parallel DO
3     IF k > 2^(j-1) THEN
4        receive(x[k − 2^(j-1)])
5        x[k] = x[k − 2^(j-1)] ⊕ x[k]
6     END IF
7   END FOR
8 END FOR
```

Algorithm 6-1 can be modified (as in algorithm 6-2) to compute the recursion for the case where the number of elements in the sequence is greater than the number of processors used by the algorithm. That is, $\mathrm{n} > \mathrm{p}$, where n is the number of elements in the input sequence as well as the output sequence and p is the number of processes used for the computations. Again, as is commonly assumed, simplifying assumptions for algorithm 6-2 are, n is divisible by p and p is an integral power of 2.

The computational runtime complexity of sequential steps 1 and 3 of algorithm 6-2 is $O(\mathrm{n}/\mathrm{p})$ and that of step 2 is $O(\log \mathrm{p})$. Hence the computational runtime complexity of algorithm 6-2 is $O(^{\mathrm{n}}/_{\mathrm{p}} + \log \mathrm{p})$. The time required for communications is same as that of step 2, as steps 1 and 3 do not require communications. The communication time complexity of algorithm 6-2 can be characterized as $O[(\gamma + \eta \mathrm{m}) \log \mathrm{p}]$. Here, γ is the

time required for setting up or initializing a communication (or access to memory in a shared memory system), η is the rate at which a unit of information is communicated, and m is the size of communication. The setup cost includes the time required to compose the message, add required headers or other information as required.

Algorithm 6-2

```
Step 1:

  Each processor computes the prefixes locally using a
  sequential algorithm.

Step 2:

  Using the last element for each of the local prefix scan
  outputs, perform Algorithm 6-1.

Step 3:

  In each processor compute the final prefix scan output by
  combining the outputs of step 1 and 2.
```

Many applications of parallel prefix, for example the sequence alignment problem, N-body simulation to name a couple, use a series of parallel prefix computations to arrive at their final result. Where a series of parallel prefixes is defined as a sequence of parallel prefix computations such that the results of one depends on the result of previous parallel prefix algorithms, which in turn might depend on results of other parallel prefixes. In this section we describe a reduction to optimize the computation by composing a series of parallel prefixes into one.

Let us begin by considering the following simple case, where the result of first parallel prefix is used as an input for the second parallel prefix computation. It can be implemented as presented in algorithm 6-3 using algorithm 6-1 or 6-2.

Algorithm 6-3

```
1  X = scan(A, ⊕)
2  Y = scan(X, ⊗)
```

From the definition presented by equation 1, both the operators $\oplus$ and $\otimes$ are binary associative. With the added constraint that $\oplus$ distribute over $\otimes$ and B = n/p, we define composite parallel prefix for Algorithm 6-3 as algorithm 6-4.

Algorithm 6-4

```
Step 1:
1 FOR all k in parallel DO
2   FOR i = (k − 1)B + 1 to kB DO
3     x[i] = x[i − 1] ⊕ a_i
4     y[i] = y[i − 1] ⊗ x[i]
5   END FOR
6   x′[k] = x[kB]
7   y′[k] = y[kB]
8 END FOR

Step 2:
1 FOR j = 1 to log2 p DO
2   FOR all k in parallel DO
3     IF k > 2^(j−1) THEN
4       receive( x′[k − 2^(j−1)], y′[k − 2^(j−1)] )
5       x′[k] = x′[k − 2^(j−1)] ⊕ x′[k]
6       y′[k] = y′[k − 2^(j−1)] ⊗ (x′[k − 2^(j−1)] ⊕ y′[k] )
7     END IF
8   END FOR
9 END FOR

Step 3:
1 FOR all k > 1 in parallel DO
2   x″[k] = x′[k − 1]
3   y[(k − 1)B] = y′[k − 1]
4   FOR i = (k − 1)B + 1 to kB − 1 DO
5     y[i] = y[i − 1] ⊗ (x″[k] ⊕ x_i)
6   END FOR
7 END FOR
```

Lines with bold font in algorithm 6-4 depict computations that are different from algorithm 6-2. Also, it must be noted that to simplify the presentation, the algorithm description introduces extra memory to store some of the intermediate results (x', y', x''). These are not required for implementation of both algorithm 6-2 and 6-4.

The key insight that makes this composition possible is that although all the $\otimes$ operations in steps 1 and 2 of algorithm 6-4 operate on partial results, the results are correct. For instance, in the case where $\oplus$ is Addition and $\otimes$ is Max operators, the results are correct because the difference of the partial results (the arguments to $\otimes$) and their respective complete results are equal. Simply put,

$$\max(x[i] + x'', x[j] + x'') = \max(x[i], x[j]) + x'' \quad (2)$$

Where $x[i]$ and $x[j]$ are the local partial results and x'' is the prefix scan result from all the previous elements globally. This property can be extended to be applicable for any number of prefix scans such that the operator properties hold, for example scan with Addition and Max operators, scan with Multiplication and Addition operators, scan on a sequence of positive rational numbers with Multiplication, Addition and Max operations. It should be noted that the order of operators is of importance, for example Multiplication distributes over Addition but Addition does not distribute over Multiplications. That is while 3 holds, 4 does not.

$$(x[i] * x'') + (x[j] * x'') = (x[i] + x[j]) * x'' \quad (3)$$

$$(x[i] + x'') * (x[j] + x'') \neq (x[i] * x[j]) + x'' \quad (4)$$

Although both, algorithm 6-3 and algorithm 6-4, have similar number of computations. Algorithm 6-3 exchanges twice as many messages when compared to algorithm 6-4. This is achieved by aggregating the messages for the prefix scans. As will be seen in later sections, this results in both execution time as well as the energy efficiency of the computation.

More formally, if K parallel prefixes are composed, although the computation time complexity remains the same, time required for communication is different. Composite parallel prefix computation requires $log\ p$ communications each of size $K\times m$, instead of $K\times \log p$ communications of size m required for K parallel prefix computations. Hence, the communication time complexity for composite parallel prefix computation is $O[(\gamma + K\eta m)\log p]$. Whereas, the communication time complexity for K consecutive parallel prefixes is $O[K(\gamma + \eta m)\log p]$. Assuming $\gamma \gg \eta$ and that the bounds are tight, the speedup for the composite parallel prefix when compared to K consecutive parallel prefixes is K.

In addition to being faster the aggregation of messages is expected to reduce the energy per bit required for communication as was seen in (Biswas and Fujimoto 2017, Biswas and Fujimoto 2018). Following the energy model introduced in section 5.5.3 and the assumption that $\tau \gg \mu$, composite parallel prefix is expected to be K times more energy efficient when compared to K parallel prefix computations.

It is also worth noting that a sequence of parallel prefix computations that does not have dependence on the output of other parallel prefix computations (i.e. if second scan in algorithm 6-3 was based on any other input other than X) then the series can be

trivially executed in an embarrassingly parallel execution. Also, the series can be composed using algorithm 6-4. Line 6 in step 2 and line 5 in step 3 can be changed to reflect the dependence on just the input sequence. Also, in this case the distributive properties of the operators can be relaxed.

6.3 Queueing Network Simulations

In this section, we first propose a recurrence for parallel and distributed queueing network simulation using data parallel algorithms. Then we describe methodologies to implement the proposed algorithm in a parallel computing environment.

Many key metrics of interest when simulating a queueing network can be derived from the departure time of each job. For example, the waiting or sojourn time of a job or group of jobs in a system of queues is simply given by the difference in a jobs arrival time and its departure times, less the service time. For an infinite queue, the trajectory of queue length or sample path can be computed by merging the sequence of arrival times and the departure times of the jobs. Both quantities can be computed efficiently (sequentially or in parallel) once the departure times are known. The arrival, service, and departure times of a job at a queue denote the simulation time at which the job arrives, the simulation time required by a server to serve the job, and the time it has completed receiving service and leaves the queue, respectively.

Kendall's notation is a common nomenclature used to describe the properties of a queue; a detailed description of the notation can be found in (Heyman 2013). Briefly, the nomenclature is represented in the form A/S/c/K/Q. Where, A refers to the inter-arrival time distribution, S the service-time distribution, c the number of parallel servers, K

maximum system size (assumed infinity is omitted) and Q the queueing discipline (assumed FCFS if omitted). In this work we focus on G/G/1 queues. A G/G/1 queue represents a First Come First Serve (FCFS) based queue with one server serving jobs. The events in a G/G/1 queue arrive with independent inter arrival times and have independent service times. In other words, the arrival time and the service times of the jobs in the queue are arbitrary. Hence, G/G/1 queues could be considered the most general case for FCFS based single-server queue. For G/G/1 queues the following linear recurrence relation provides an efficient sequential method to compute the departure time, D_i for i^{th} job in the queue.

$$A_i = A_{i-1} + \alpha_i \tag{5}$$

$$D_i = \max\begin{Bmatrix} A_i \\ D_{i-1} \end{Bmatrix} + \delta_i \tag{6}$$

Where, α_i is the inter-arrival time, A_i is the arrival time of the i^{th} job and δ_i is the required service time for the units of time of service. $A_0 = D_0 = 0, \forall\, i \geq 1$ and $\max\begin{Bmatrix} x \\ y \end{Bmatrix}$ represents MAX operation i.e. it denotes maximum of x and y.

It should be noted that all the parallel implementations presented and discussed in the following sections are exact simulations (as opposed to approximate). In other words, the result of the parallel simulation matches with that of a corresponding sequential model.

6.3.1 Recurrence for G/G/1 Queues

We start with the linear recurrence in equation 6 and begin by rearranging the service time.

With $A_0 = D_0 = \delta_0 = 0 \,\&\, i \geq 1$ we can rewrite it as follows:

$$D_i = \max\begin{Bmatrix} A_i \\ D_{i-1} \end{Bmatrix} + \delta_i$$

$$D_i - \delta_i = \max\begin{Bmatrix} A_i \\ D_{i-1} \end{Bmatrix}$$

$$D_i - \delta_i - \sum_{j=0}^{i-1} \delta_j = \max\begin{Bmatrix} A_i - \sum_{j=0}^{i-1} \delta_j \\ D_{i-1} - \sum_{j=0}^{i-1} \delta_j \end{Bmatrix}$$

$$D_i - \sum_{j=0}^{i} \delta_j = \max\begin{Bmatrix} A_i - \sum_{0}^{i-1} \delta_j \\ D_{i-1} - \sum_{0}^{i-1} \delta_j \end{Bmatrix}$$

Rearranging the service time and subtracting exclusive cumulative sum of the service times, results in the form above. The apparent, a pattern among the terms on the LHS and the terms in the second argument of the max function can be leveraged to convert the resultant form into a recurrence. Let, $K_i = D_i - \sum_{j=0}^{j=i} \delta_j$. We get the equation 7 by replacing the terms with K_i and K_{i-1}.

$$K_i = \max\begin{Bmatrix} A_i - \sum_{j=0}^{i-1} \delta_j \\ K_{i-1} \end{Bmatrix} \quad (7)$$

Following from equation 5, the arrival times A_i in equation 7 can be re-written as a cumulative sum of the inter-arrival times. That is, $A_i = \sum_{j=0}^{j=i} \alpha_j$. Substituting this in equation 7, we get equation 8.

$$K_i = \max_{K_{i-1}} \left\{ \sum_{j=0}^{j=i} \alpha_j - \sum_{j=0}^{j=i-1} \delta_j \right\} \quad (8)$$

As will be seen in the next sub-section, the linear recurrence presented by equation 7 and 8 can be implemented using parallel prefix scan.

6.3.2 Implementation Starting from Arrival Times

A naïve implementation of the recurrence can be achieved using the following algorithm.

Algorithm 6-5: naïve implementation for G/G/1 queues starting from arrival times

```
Step 1: δ' = ExclusiveScan(δ, +)
Step 2: A′ = A − δ′
Step 3: K  = InclusiveScan(A′, MAX)
Step 4: D  = K + δ′ + δ
```

Here an array of values composed of X_i is represented as X. ExclusiveScan(X, +) represents exclusive scan with addition operations on elements in X; similarly InclusiveScan(X, MAX) represents inclusive scan with the MAX operation on elements of X. Also, all the array operations are element wise operations (steps 2 and 4). As steps 2 and 4 of the algorithm can be computed in embarrassingly parallel manner, it can be noted that algorithm 6-5 can be computed in parallel, possibly even with a distributed memory architecture, if an appropriate implementation of the scan primitive is used in steps 1 and 3.

In terms of time complexity, steps 1 and 3 can be computed in O(N/P + log P) i.e. the time complexity of a parallel prefix scan of N elements on a system with P processing

elements and steps 2 and 4 in O(N/P), resulting a asymptotic runtime of O(N/P + log P). The space complexity of the proposed algorithm is also O(N) but can be implemented with O(1) auxiliary space complexity.

6.3.2.1 Composite Parallel Prefix Implementation

The structure of equation 7 allows a more optimized implementation. This optimized implementation is achieved by combining the two parallel prefix scan operations into one composite parallel prefix scan implementation because addition distributes over max operation. Algorithm 6-6 discusses a composite parallel prefix based implementation for equation 7.

The implementation is discussed assuming $N \geq P$, and that N is divisible by P and $N/P = B$. Where B is the size of a block of data associated with a processing element and assume a distributed memory machine. For a shared memory machine, algorithm 6-6 can be implemented by changing message exchanges to memory accesses in step 2. The divisibility assumption can be easily relaxed by padding the input.

It can be observed that first and third lines of step 1, are exclusive prefix scans with addition operation and inclusive scans with max operation, respectively. For simplicity algorithm 6-6 shows these as repeated scans over the arrays but each step of the prefix scans can be computed using the result of the previous step, which is locally available. Hence, the steps can be computed in one pass over the array.

Algorithm 6-6

In parallel, let each processor P_x (where $k \in [1, P]$) locally and sequentially compute:

Step 1:

1 $\delta'_i = \sum_{j=(k-1)B+1}^{j=i-1} \delta_j \quad \forall i \in [(k-1)B+1, kB]$

2 $A'_i = A_i - \delta'_i \quad \forall i \in [(k-1)B+1, kB]$

3 $K'_i = \max \begin{Bmatrix} A'_i \\ A'_{i-1} \end{Bmatrix} \quad \forall i \in [(k-1)B+2, kB]$

and $A_{(k-1)B+1} = \delta'_{(k-1)B+1} = \delta_{(k-1)B+1}$

Step 2:

```
1    L_k = K'_kB
2    Δ_k = δ'_kB
3    FOR i = 0 to log p − 1 DO:
4        k' = (k − 1) XOR 2^i
5        IF k − 1 > k' THEN:
6            receive(L_k', Δ_k')
7            L_k = max (L_k − Δ_k', L_k')
8            Δ_k = Δ_k + Δ_k'
9        END IF
10    END FOR
12   END FOR
13
14   IF x > 1 THEN:
15    receive(L_k−1, Δ_k−1)
16    L_K = L_k−1
17    Δ_k = Δ_k−1
18   ELSE:
19    L_K = −∞
20    Δ_k = 0
21   END IF
```

Step 3:

1 $D_i = \max \begin{Bmatrix} A'_i - \Delta_k \\ L_k \end{Bmatrix} + \Delta_k + \delta'_i \quad \forall i \in [(k-1)B+1, kB]$

Each processor locally computes the partial exclusive prefix scan of service times with addition operations. The algorithm then computes $A_i^{'}$ using the inclusive prefix scan of $A_i^{'}$ with MAX operations. Step 2 combines these results across processors returning

two exclusive prefix scan results, which are then combined with results based on partial prefix scan to generate the final result.

As before, step 2 implements a variant of parallel prefix scan as proposed by Hillis and Steele (Hillis and Steele Jr 1986) but can be extended to other parallel prefix algorithms. In step 2, it can be inferred that the output values Δ_k and L_k are respectively results of exclusive scan of service times and K' for respective array elements.

6.3.3 Implementation Starting from Inter-arrival Times

A naïve implementation of the recurrence presented by equation 8 can be achieved using algorithm 6-7.

Algorithm 6-7: naïve implementation for G/G/1 queues starting from inter-arrival times

```
Step 1: δ' = ExclusiveScan(δ, +)
Step 2: A  = InclusiveScan(α, +)
Step 3: A′ = A − δ′
Step 4: K  = InclusiveScan(A′, MAX)
Step 5: D  = K + δ′ + δ
```

As in the case of algorithm 6-5, for an array of values composed of X_i, represented as X, ExclusiveScan(X, +) represents exclusive scan with addition operations on elements in X; similarly InclusiveScan(X, MAX) represents inclusive scan with the MAX operation on elements of X. Again, the operations in steps 2 and 4 are element wise operations and hence can be computed in embarrassingly parallel manner. A composite parallel prefix version of Algorithm 6-7 can be achieved by extending Algorithm 6-6 to include the extra prefix sum.

6.4 Empirical Evaluation

In this section we discuss the empirical evaluation that was carried out to evaluate the performance of the composite parallel prefix for parallel simulation of queues. The goals of these evaluations succinctly stated are threefold. First, evaluate the advantage of queueing network simulation on GPU. For this we determine the speedup of parallel implementations with respect to each other (a direct implementation of algorithm 6-5 and 6-7 using a sequence of parallel prefixes or the naïve approach and their composite parallel prefix based versions), as well as with respect to a sequential implementation. Second, compare the execution times of composite parallel prefix based implementation and the fastest possible (library-based) implementation of the naïve approach. Third, characterize the energy consumption of composite parallel prefix and compare with library-based implementation of the naïve approach. For the final goals, a composite parallel prefix based implementation (algorithm 6-6) was compared with a naïve approach (algorithm 6-5) based implementation using NVIDIA's CUB library primitives. NVIDIA's CUB framework provides highly optimized scan operations and is arguably the fastest one currently available. These primitives outperform primitives implemented by other libraries like Thrust.

Next, we briefly discuss the implementations developed for experimentation and the platform of experimentation. For each of the implementations discussed in the following subsections, the individual queuing simulation phase was combined with a parallel sorting phase to simulate a corresponding closed 2D torus network.

6.4.1 Sequential Implementation

The sequential implementation serves two purposes, one for verification of the results of the parallel implementations and second as a baseline for comparison and speed up computations.

The sequential implementation processes each job sequentially and utilizes the recurrence mention in equation 6 for efficiently computing the departure time of the job. Algorithm 6-8 provides a pseudo code for the sequential implementation. A variation of this implementation was developed for simulating queues starting from inter-arrival times.

Algorithm 6-8: Sequential computation of departure times for G/G/1 queues starting from arrival times

```
1 FOR each queue q DO
2    D[q, 0] = A[q, 0] + δ[q, 0]
3    FOR each i = 1 to queue_length DO
4       D[q, i] = max(A[q, i], d[q, i − 1]) + δ[q, i])
5    END FOR
6 END FOR
```

6.4.2 Parallel Implementations

Five parallel implementations were developed for experimentation. First, the implementation of algorithm 6-5 is referenced as the '*naïve*' implementation. Second, the implementation of algorithm 6-7 is called '*naïve starting from inter-arrival time*'. Third, the implementation of algorithm 6-6 is the '*composite parallel prefix'* implementation. Fourth, a composite parallel prefix implementation of algorithm 6-7 was developed, referenced as the '*composite parallel prefix starting from inter-arrival time*'. All the aforementioned implementations use a variation of scan code included as part of the CUDA Toolkit. Finally, an implementation of algorithm 6-5 was developed using scan

primitives of CUB library. This will be referenced as the '*library based naïve*' implementation. The outputs of all the implementations are checked to be exactly same as the corresponding sequential implementations and hence are same for corresponding parallel implementations for a given input. In other words, all the implementations sequential and parallel, starting from inter-arrival times have same output and all the ones starting from arrival times have same output.

6.4.3 Platform of Experimentation

The sequential runs were conducted using one process on an Intel® Xeon® CPU E5-2623 v4 with a maximum frequency of 2.60GHz. The parallel implementations were executed on a NVIDIA Tesla P100 GPU, with a host CPU same as in the case of sequential. The energy values were computed as the product of the execution time and average power. Average power was computed from instantaneous power values observed using nvidia-smi tool at a frequency of 1Hz.

6.5 Empirical Results

In this section we present the results of the empirical studies conducted using the implementations discussed in the previous section. Two sets of experiments were conducted. For the first set or micro-benchmarking experiments, 512 separate finite queues were considered. For the second set of experiments finite queues were arranged to form a closed 2D torus network of dimensions 32x16. For the second set of experiments, the simulations progressed in a barrier based manner, or epochs. Although we use a 2D torus queueing network, it should be noted that the recurrences derived earlier could be used with any queuing network architecture. The first set of experiments was targeted at

the first goal, mentioned in the beginning of last section, while the second set targeted the second and third goals. The micro-benchmark evaluates the efficiency of composition over direct implementation of the parallel algorithms. A network of queues provides a more realistic scenario and is applicable to a very wide range of applications and hence was used for the second set.

In both sets of experiments, the queues were initialized with synthetically generated random arrival times in increasing order or synthetically generated inter-arrival times at the beginning of the simulation. The service times were randomly assigned for each job and were synthetically generated as well. All the runtime statistics, i.e. execution time and power, were computed as the average over 1000 runs. The speedups and energy improvements were computed using 7 such execution time and energy values, respectively. The current implementation of the torus network does not lose any jobs during the simulation, i.e. the total number of jobs in the network stays constant throughout the simulation.

6.5.1 Micro-benchmarking Experiments

Figure 6-1 plots the speed up of the parallel implementation with respect to the sequential implementation. The speedups for both composite and naïve implementations increase with the number of jobs in the queue. Assuming a linear speedup for a distributed simulation on multiple CPUs for the queueing network, a similar ratio is expected in terms of energy consumed by the GPU vs CPU implementations. More interestingly, figure 6-2 plots speedup for the *composite* with respect to naïve (green) and *composite starting from inter-arrival* with respect to *naïve starting from inter-arrival* (orange).

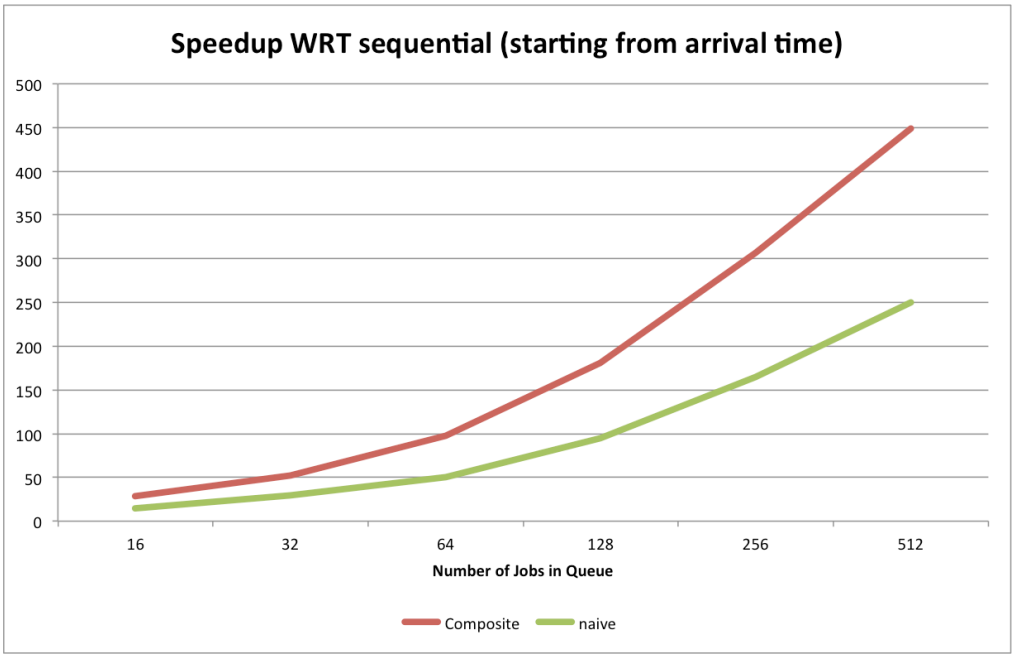

Figure 6-1: Comparison of Speedup for parallel implementation with respect to the sequential implementation.

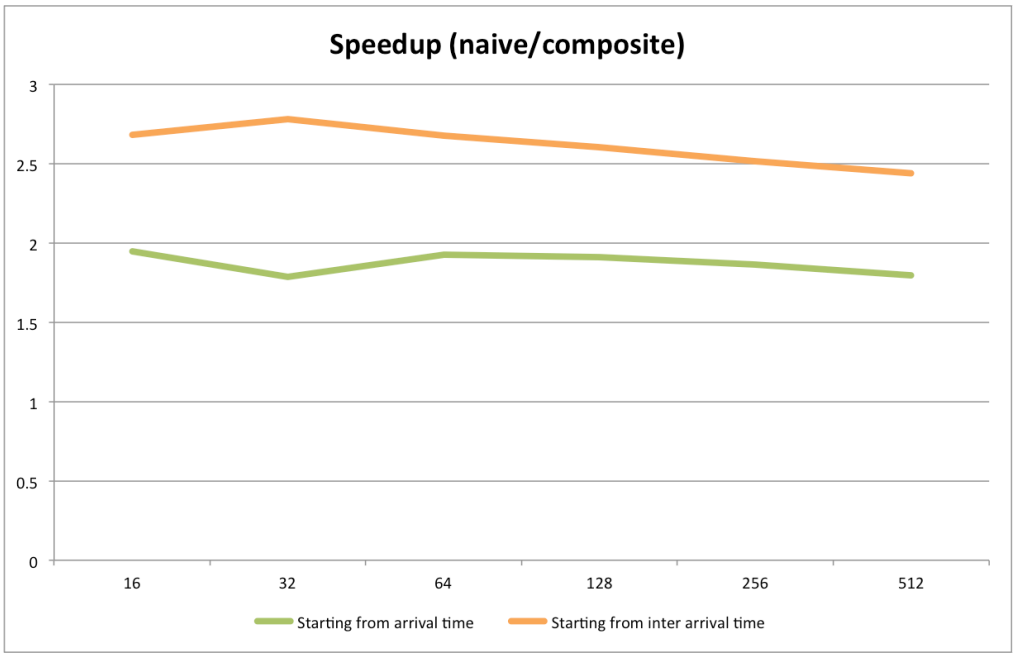

Figure 6-2: Variation of speedup for the *composite* versions with respect to their *naïve* counterparts as the number of jobs in queues varies. Observed speedup confirms the expectation that speedup is approximately k when k scans are composed.

The composite implementation in the first case composes 2 parallel prefixes. It has an average speedup of 1.9. In the second case, where 3 parallel prefixes were composed, the average speedup is 2.6. According to the analysis presented in the later part of section 6.2, the expected speedup for first and second cases would be 2 and 3 respectively. The decay in the speedup can be attribute to the increasing ratio of the computation versus communication time.

6.5.2 Torus Network Experiments

Two properties of the queueing network were varied for the experiments presented in this section. The first is the number of jobs each queue, and hence the network. The second parameter is the number of epochs for which the simulation is conducted. In terms of implementations, the composite and the library based naïve approaches are studied in this section. In the following subsections first, we compare the execution time of the torus network implementations and then their power and energy characteristics. As will be seen shortly, *composite* implementation is faster and more energy efficient than the *highly optimized library based naïve* implementation. This points to the fact that an optimized version of the *composite* will perform even better.

6.5.2.1 Comparing Execution Time

With the number of epochs fixed to 1000, figure 6-3 presents the comparison for execution time for increasing number of jobs in the queue. Although the execution time increases with the increase in number of jobs for both the implementations, the increase is faster for the *library-based naïve*. *Composite* implementation executes faster in all cases, with speedup ranging from 1.1 for 16 jobs to 1.3 for 1024 jobs per queue.

The next set of experiments fixed the number of jobs in the queue to 512, the results of the increasing number of epochs is presented in figure 6-4. Like the previous case, composite is faster in all cases. But unlike previous case, the speedup stays very stable, around 1.29.

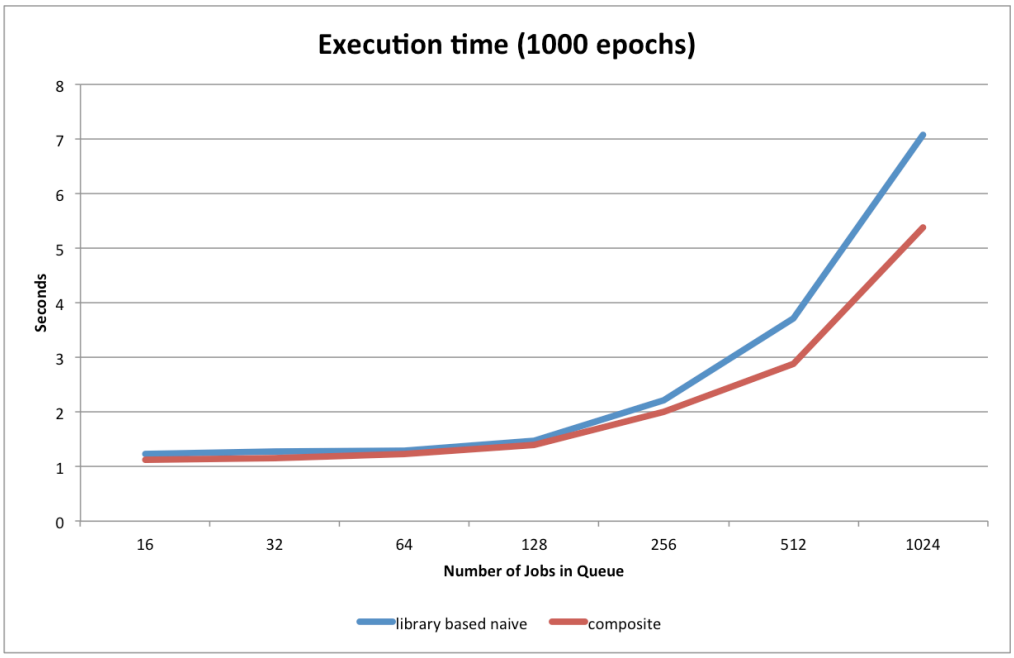

Figure 6-3: Speedup of composite with respect to *library based naïve* in general increases (from 1.1 to 1.31) with increasing number of jobs.

6.5.2.2 Comparing Energy Consumption Behavior

Instantaneous power was monitored for the experiments described earlier in this section. Figure 6-5 plots the average power consumed by the implementations for an increasing number of jobs per queue. Although for smaller queues *composite* performs marginally better than the *library based naïve* implementation, it quickly supersedes it. This increase in power for the larger queue sized can be attributed to increase in size of arrays being handled.

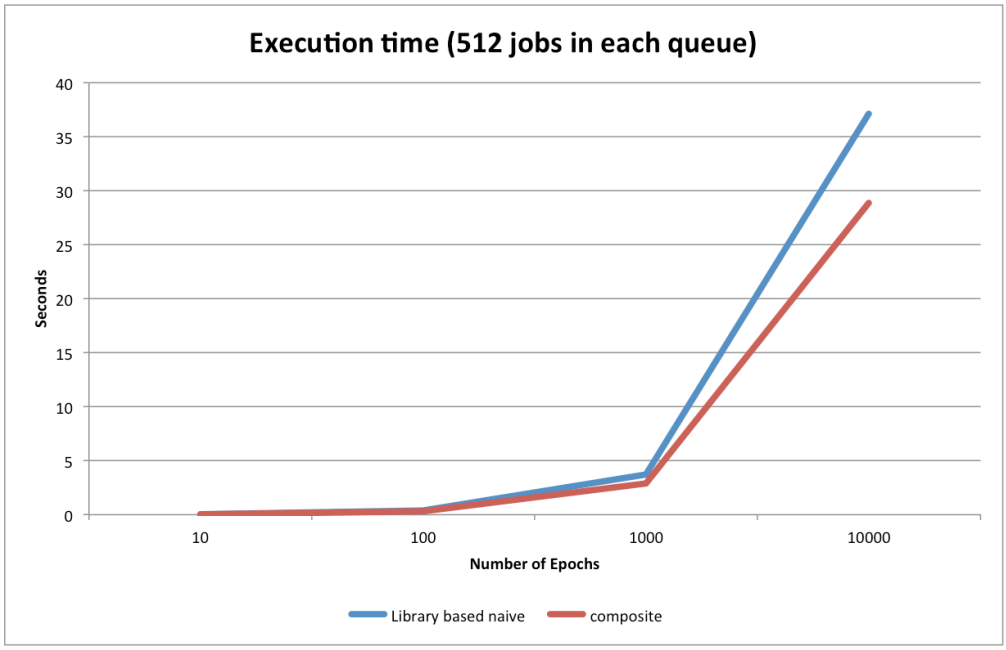

Figure 6-4: Speedup of *composite* with respect to *library based naïve* stays stable at approximately 1.29 for increasing number of epochs.

Energy savings are computed as the difference in energy required for the execution of library based naïve implementation and composite implementation. The energy consumed was computed as the product of the average power and execution time. As depicted by the Figures 6-6 and 6-7, the advantage of composition in the execution time beats the disadvantage in power. Hence in all cases the *composite* is more energy efficient than *library based naïve* implementation. These savings can be described as the reduction in energy required per unit data communicated, due to message aggregation.

6.6 Discussion

Although our empirical analysis, used for demonstrating the advantages of the composite parallel prefix, uses shared memory SIMD based systems, all proposed algorithms and analysis can be carried over to a distributed memory systems as well. For

a distributed memory system, the advantages of the composition are expected to be more profound, as the communication costs are much higher.

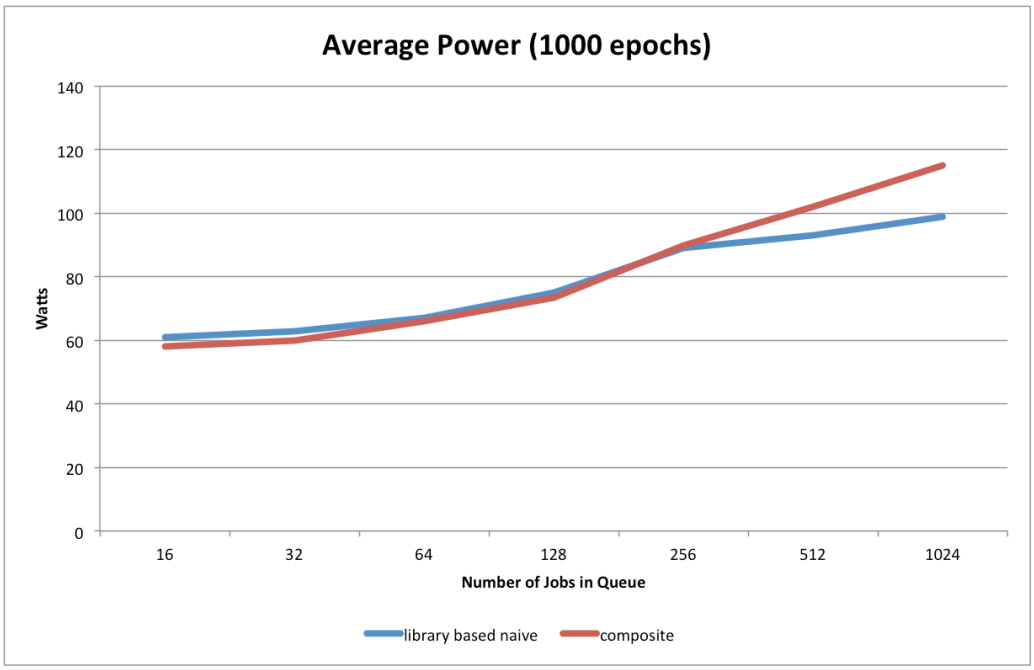

Figure 6-5: *Library based naïve* is more power efficient than composite.

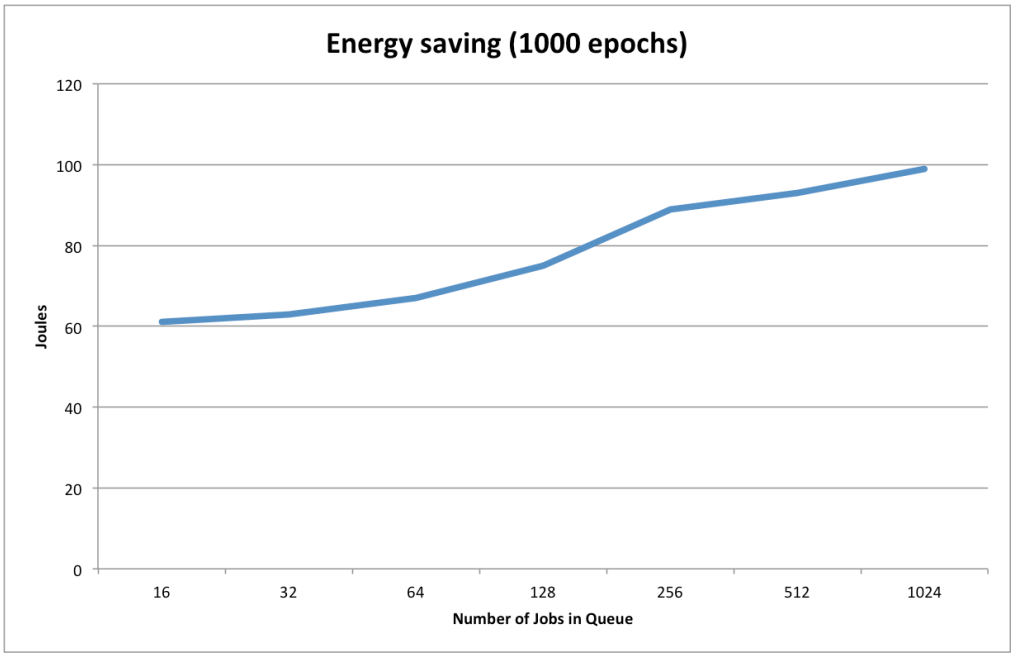

Figure 6-6: Energy savings for the *composite* with respect to the *library based naïve* implementation increases with the number of jobs in the network.

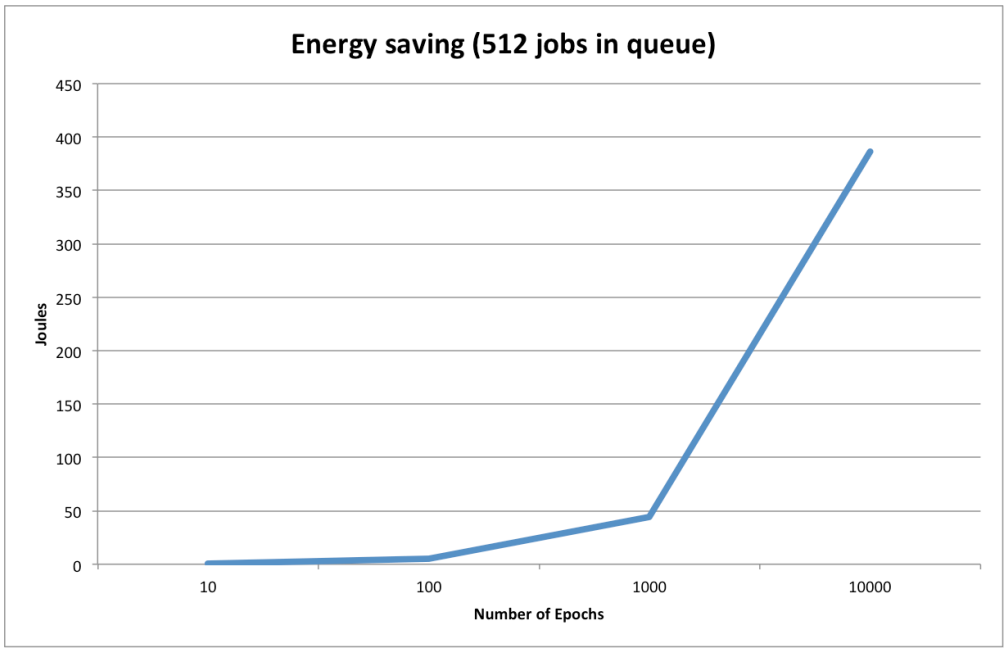

Figure 6-7: Energy savings for *composite* when compared to *library based naïve* implementation increases linearly with the increase in number of epochs.

Energy and power characteristics of the implementations observed in section 6.2.2 are interesting. In addition to portraying the energy efficiency of the composition, it also suggests that although the *composite* implementation is better suited in general, the *library based naïve* implementation would perform better in a power-capped environment.

It is increasingly common for hardware to be optimized for commonly used computing primitives, such as Tensor Processing Unit (TPU) for matrix multiplication. Given the generic and simple nature of the operations of the recurrence derived for the queueing simulation, if such optimizations are available for the primitives, the derived recurrence in equation 7 and 8 would outperform other similar queueing simulation approaches that use more idiosyncratic operations. In addition to this property, the derived recurrence provides a higher degree of parallelism than that of traditional PDES. Degree of parallelism in traditional PDES is generally restricted by the number of queues.

The proposed recurrence also lends to optimizations like composition and the special case of G/D/1 queues.

6.6.1 Special Case: Recurrence G/D/1 Queue Simulation

A special case of G/G/1 queue is when the service times of the jobs arriving in the queue are deterministic. By implication of the deterministic property of the service times, we have

$$\delta_i = \delta \qquad \forall\, i \geq 1$$

$$\therefore \sum_{j=0}^{j=X} \delta_j = X\delta \tag{9}$$

∴ Starting from equation 7 and substituting equation 9, we can simplify equation 6 for this special case as equation 10.

$$K_i = \max\begin{Bmatrix} A_i - (i-1)\delta \\ K_{i-1} \end{Bmatrix} \tag{10}$$

Where, $K_i = D_i - i\delta, A_0 = D_0 = \delta_0 = 0\ and\ i \geq 1$. This in turn simplifies Algorithm 6-5. Which can be re-written, such that K can be computed using a single parallel prefix pass.

6.7 Conclusions

In this chapter we proposed a recurrence relation for simulation of FCFS based G/G/1 queues. We also discussed an optimization of the proposed recurrence using composite parallel prefix. Composition of parallel prefixes is defined as an optimization for any series of parallel prefix computations, where the operation of a parallel prefix distributes over the operation of the subsequent parallel prefixes in the series. Here a

series of parallel prefix scans is defined as a sequence of parallel prefix computations such that the results of one depends on the result of previous parallel prefix algorithms, which in turn might depend on results of other parallel prefixes. We present an analysis and discuss the performance improvements introduced by composition and empirically support the claims. For the empirical analysis, implementations of the proposed data parallel recurrences for simulating G/G/1 queues were used. The analysis shows that the proposed recurrences are significantly faster and more energy efficient than the sequential approaches and that the composition of parallel prefixes has advantages in terms of execution time as well as energy.

7 CONCLUSION AND FUTURE WORK

Energy and power have evolved to be a primary concern for computing. For larger computing platforms such as datacenters and super computers energy translates into financial costs and an enormous carbon footprint. For mobile systems such as edge computing platforms energy consumption can restrict the range of applications suitable for the platform. Computing on both of these systems is increasing in importance. On the one hand the increasing computational power and networking capabilities at the edge ensure increased ubiquity of mobile computing devices. On the other hand, the next generation of supercomputers, with exa-scale computing capabilities are on the horizon. These form opportune conditions for computationally intensive software applications such as simulation to expand its usage and significance in existing and emerging computing platforms.

This thesis makes a case for efficiency of the software component of a parallel and distributed simulation system. Underlining concerns are the significance of computationally efficient algorithms, in addition to efficient hardware, for improving energy efficiency of the system.

This work explored the energy and power consumption characteristics of distributed simulations, more specifically parallel and distributed discrete event simulations synchronized with conservative synchronization algorithms, and area that has received only limited attention by other researchers. Some of the major conclusions of these studies were:

1) The choice of synchronization algorithms can greatly affect the energy consumption of the distributed simulations.
2) The overhead of distribution can be significant in terms of energy.

Developing on these, the next part of the work focused on improving the energy efficiency of simulations. Some of the major conclusions of this part of the work were:

1) Efficient and dynamic middlewares can ease the development of efficient DDDAS systems.
2) Compared to classical synchronization algorithms, ones designed for energy efficiency can significantly improve the energy efficiency of distribution simulation systems.
3) By designing energy efficient synchronization algorithms, it is possible for a distributed simulation to approach its theoretical minimum in terms of energy consumption.
4) Use of application specific simulation algorithms and non-traditional hardware platforms such as GPUs can dramatically improve the energy efficiency.

In terms of future directions, further improvement of efficiency and applicability of composition of parallel prefix, as discussed in section 6.2, will be an advantageous extension of this work. One possible avenue for improvement could be to reduce the message size by eliminating or reducing partial results exchanged among the processes.

Furthermore, studies in this work primarily focused on conservative synchronization algorithms. The characteristics and optimization of the optimistic synchronization algorithms was not addressed in this work. A similar study of distributed

simulations synchronized with optimistic synchronization algorithms, using the tools and models developed in this study, is a possible future direction of research. Such studies will provide insight into the characteristic differences between the two classes of synchronization algorithms, allowing application developers to choose the ones best suited for the application at hand.

Finally, approaches to dynamically change the behavior of the distributed simulation based on the energy available in batteries or based on the power being consumed on a high performance computing platform, also have not been explored in this work. Power- and energy-aware parallel and distributed simulation represents a rich area of possible future research with many unsolved problems.

REFERENCES

Aberer, K., M. Hauswirth and A. Salehi (2006). The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks.

Acun, B., A. Langer, E. Meneses, H. Menon, O. Sarood, E. Totoni and L. V. Kalé (2016). "Power, reliability, performance: One system to rule them all." Computer **49**(10): 30--37.

Ang, J., K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, A. Geist and G. Grider (2014). Top ten exascale research challenges. Advanced Scientific Computing Advisory Committee, US Department Of Energy Report.

Bard, Y. (1977). The Modelling of Some Scheduling Strategies for and Interactive Computer System. Computer Performance, Elsevier North-Holland, Inc.

Bedard, D., R. Fowler, M. Y. Lim and A. Porterfield (2009). PowerMon 2: Fine-grained, Integrated Power Measurement RENCI Technical Report TR-09-04.

Bhatti, K., C. Belleudy and M. Auguin (2010). Power management in real time embedded systems through online and adaptive interplay of DPM and DVFS policies. International Conference on Embedded and Ubiquitous Computing. Hong Kong, China, IEEE**:** 184–191.

Biswas, A. and R. Fujimoto (2016). Profiling Energy Consumption in Distributed Simulations. ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS), Banff, Alberta, Canada, ACM.

Biswas, A. and R. Fujimoto (2017). "Energy consumption of synchronization algorithms in distributed simulations." Journal of Simulation **11**(3): 242--252.

Biswas, A. and R. Fujimoto (2018). Zero Energy Synchronization of Distributed Simulations. ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, Rome, Italy, ACM.

Biswas, A., M. Hunter and R. Fujimoto (2018). Energy Efficient Middleware For Dynamic Data Driven Application Systems. Winter Simulation Conference (WSC) 2018. M. Rabe, A. A. Juan, N. Mustafee et al. Gothenburg, Sweden, IEEE**:** 628--639.

Blasch, E., S. Ravela and A. Aved (2018). Handbook of dynamic data driven applications systems, Springer International Publishing.

Blelloch, G. E. (1989). "Scans as primitive parallel operations." IEEE Transactions on computers **38**(11): 1526--1538.

Blelloch, G. E. (1990). Prefix sums and their applications, Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University,.

Blelloch, G. E. (1990). Vector models for data-parallel computing, MIT press Cambridge.

Bornholt, J., T. Mytkowicz and K. S. McKinley (2012). The model is not enough: Understanding energy consumption in mobile devices. IEEE Hot Chips 24 Symposium (HCS), Cupertino, CA, USA, IEEE.

Bryant, R. E. (1977). Simulation of Packet Communication Architecture Computer Systems M.S. thesis, MIT-LCS-TR-188, Massachusetts Institute of Technology.

Cardoso, J., Pereira, C., Aguiar, A., & Morla, R. (2017). Benchmarking IoT middleware platforms. IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks IEEE**:** 1-7.

Casbeer, D. W., R. W. Beard, T. W. McLain, S.-M. Li and R. K. Mehra (2005). Forest fire monitoring with multiple small UAVs. Proceedings of the American Control Conference.**:** 3530--3535.

Chandy, K. and J. Misra (1981). Asynchronous distributed simulation via a sequence of parallel computations. Communications of the ACM. **24**.

Chandy, K. M. and J. Misra (1979). "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs." IEEE Transactions on Software Engineering **SE-5**(5): 440-452.

Child, R. and P. A. Wilsey (2012). Using DVFS to optimize time warp simulations. Winter Simulation Conference.

Chiu, W. W., D. Dumont and R. Wood (1975). "Performance analysis of a Multiprogrammed Computer System." IBM J. of Research and Developement **19**(3): 263-271.

Cho, K.-M., C.-H. Liang, J.-Y. Huang and C.-S. Yang (2011). Design and implementation of a general purpose power-saving scheduling algorithm for embedded systems. IEEE International Conference on Signal Processing, Communications and Computing. Xi'an, China, IEEE**:** 1–5.

Cronin, B. (2012). "Vehicle Based Data and Availability." Retrieved 1 June, 2019, from http://www.its.dot.gov/itspac/october2012/PDF/data_availability.pdf.

Cugola, G., H. Jacobsen and others (2002). "Using publish/subscribe middleware for mobile systems." ACM SIGMOBILE Mobile Computing and Communications Review **6**(4): 25-33.

Czechowski, K. and R. Vuduc (2013). A Theoretical Framework for Algorithm-Architecture Co-design. IEEE International Symposium on Parallel Distributed Processing. Boston, MA, IEEE**:** 791-802.

Darema, F. (2004). Dynamic Data Driven Applications Systems: A New Paradigm for Application Simulations and Measurements. International Conference on Computational Science. Kraków, Poland, Springer**:** 662-669.

Dong, M. and L. Zhong (2011). Self-constructive high-rate system energy modeling for battery-powered mobile systems. 9th international conference on Mobile systems, applications, and services, ACM.

Dongarra, J., H. Ltaief, P. Luszczek and V. M. Weaver (2012). Energy Footprint of Advanced Dense Numerical Linear Algebra using Tile Algorithms on Multicore Architecture. The 2nd International Conference on Cloud and Green Computing.

Eichler, S., B. Ostermaier, C. Schroth and T. Kosch (2005). Simulation of car-to-car messaging: Analyzing the impact on road traffic. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, IEEE.

Esmaeilzadeh, H., T. Cao, X. Yang, S. M. Blackburn and K. S. McKinley (2012). "Looking back and looking forward: power, performance, and upheaval." Communications of the ACM **55**(7): 105--114.

Feng, X., R. Ge and K. W. Cameron (2005). Power and Energy Profiling of Scientific Applications on Distributed Systems. Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)**:** 34.

Flynn, M. J. (1972). "Some Computer Organizations and Their Effectiveness." IEEE Transactions on Computers **C-21**(9): 948 - 960.

Freeh, V. W., D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree and M. E. Femal (2007). "Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications." IEEE Trans. Parallel Distrib. Syst. **18**(6): 835--848.

Fujimoto, R. (1989). "Performance Measurements of Distributed Simulation Strategies." Transactions of the Society for Computer Simulation **6**(2): 89-132.

Fujimoto, R. (1990). "Parallel Discrete Event Simulation." Communications of the ACM **33**(10): 30-53.

Fujimoto, R. (2000). Parallel and Distributed Simulation Systems, Wiley Interscience.

Fujimoto, R. (2017). Power Consumption in Parallel and Distributed Simulations. Winter Simulation Conference.

Fujimoto, R., J. Barjis, E. Blasch, W. Cai, D. Jin, S. Lee and Y.-J. Son (2018). Dynamic Data Driven Application Systems: Research Challenges and Opportunities. Winter Simulation Conference (WSC), Gothenburg, Sweden, IEEE.

Fujimoto, R. and A. Biswas (2015). An empirical study of energy consumption in distributed simulations. 19th International Symposium on Distributed Simulation and Real Time Applications, China, IEEE.

Fujimoto, R. and A. Biswas (2015). An Empirical Study of Energy Consumption in Distributed Simulations. IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications.

Fujimoto, R., R. Guensler, M. Hunter, H.-K. Kim, J. Lee, J. Leonard III, M. Palekar, K. Schwan and B. Seshasayee (2006). Dynamic Data Driven Application Simulation of Surface Transportation Systems. Workshop on Dynamic Data Driven Application Simulations.

Fujimoto, R., A. Guin, M. Hunter, H. Park, G. Kanitkar, R. Kannan, M. Milholen, S. Neal and P. Pecher (2014). "A dynamic data driven application system for vehicle tracking." Procedia Computer Science **29**: 1203--1215.

Fujimoto, R., M. Hunter, A. Biswas, M. Jackson and S. Neal (2017). Power Efficient Distributed Simulation. ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, Singapore, Republic of Singapore, ACM.

Fujimoto, R., M. Hunter, J. Sirichoke, M. Palekar, H.-K. Kim and W. Suh (2007). Ad Hoc Distributed Simulations. Principles of Advanced and Distributed Simulation. San Diego, CA, IEEE**:** 15-24.

Garzón, E. M., J. J. Moreno and J. A. Martínez (2017). "An approach tooptimise the energy efficiency of iterative computation on integratedgpu–cpu systems." The Journal of Supercomputing **73**(1): 114--125.

Ge, R., X. Feng and K. W. Cameron (2005). Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. Proceedings of the 2005 ACM/IEEE conference on Supercomputing. Washington, DC, USA, IEEE Computer Society**:** 34--.

Ge, R., X. Feng, S. Song, H.-C. Chang, D. Li and K. W. Cameron (2010). "PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications." IEEE Transactions on Parallel and Distributed Systems (TPDS) **21**(5): 658-671.

Gonzalez, R. and M. Horowitz (1996). "Energy Dissipation in General Purpose Microprocessors." IEEE Journal of Solid-State Circuits **31**(9): 1277-1284.

Grasso, I., P. Radojkovic, N. Rajovic, I. Gelado and A. Ramirez (2014). Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU. IEEE International Parallel and Distributed Processing Symposium. Phoenix, AZ, IEEE**:** 123-132.

Greenberg, A., B. Lubachevsky and I. Mitrani (1990). Unboundedly Parallel Simulations Via Recurrence Relations. ACM SIGMETRICS conference on Measurement and modeling of computer systems

Guensler, R., H. Liu, Y. Xu, A. Akanser, D. Kim, M. P. Hunter and M. O. Rodgers (2017). "Energy Consumption and Emissions Modeling of Individual Vehicles." Transportation Research Record: Journal of the Transportation Research Board(2627): 93-102.

Heidelberger, P. and D. M. Nicol (1993). "Conservative parallel simulation of continuous time Markov chains using uniformization." IEEE Transactions on Parallel and Distributed Systems **4**(8): 906--921.

Heinzelman, W. B., A. L. Murphy, H. S. Carvalho and M. A. Perillo (2004). "Middleware to support sensor network applications." IEEE network **18**(1): 6-14.

Hennessy, J. L. and D. A. Patterson (2018). Turing Lecture 2018: A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development. https://iscaconf.org/isca2018/turing_lecture.html. Los Angeles, ACM.

Heyman, D. P. (2013). Queueing Theory. Encyclopedia of Operations Research and Management Science. S. I. Gass and M. C. Fu. Boston, MA, Springer US**:** 1234-1244.

Hillis, W. D. and G. L. Steele Jr (1986). "Data parallel algorithms." Communications of the ACM **29**: 1170--1183.

Hoeller, A., L. Wanner and A. Fröhlich (2006). A hierarchical approach for power management on mobile embedded systems. From Model-Driven Design to Resource Management for Distributed Embedded Systems**:** 265–274.

Hua, S. and G. Qu (2003). Approaching the Maximum Energy Saving on Embedded Systems with Multiple Voltages. IEEE/ACM International Conference on Computer-Aided Design**:** 26.

Huang, Y., M. Hunter, C. Alexopoulos and R. Fujimoto (2010). Ad Hoc Distributed Simulation of Queueing Networks. Principles of Advanced and Distributed Simulations.

Huang, Y.-L., R. Fujimoto, C. Alexopoulos and M. Hunter (2013). On the transient response of open queueing networks using ad hoc distributed simulations. Winter Simulation Conference, IEEE.

Hunter, M., A. Biswas, B. Chilukuri, A. Guin, R. Fujimoto, R. Guensler, J. Laval, H. Liu, S. Neal, P. Pecher and M. Rodgers (2018). Energy-Aware Dynamic Data-Driven Distributed Traffic Simulation for Energy and Emissions Reduction. Handbook of Dynamic Data Driven Applications Systems. E. Blasch, S. Ravela and A. Aved, Springer, Cham**:** 467--487.

Hunter, M., R. Fujimoto, W. Suh and H. Kim (2006). An Investigation of Real-Time Dynamic Data Driven Transportation Simulation. Winter Simulation Conference.

Hunter, M., H. Kim, W. Suh, R. Fujimoto, J. Sirichoke and M. Palekar (2009). "Ad Hoc Distributed Dynamic Data-Driven Simulations of Surface Transportation Systems." Transactions of the Society for Modeling and Simulation Intl. **85**(4): 243-255.

Jefferson, D. (1985). "Virtual Time." ACM Transactions on Programming Languages and Systems **7**(3): 404-425.

Jefferson, D. and H. Sowizral (1982). Fast concurrent simulation using the time warp method, Part I: Local control. Santa Monica, California, The RAND corporation.

Kamal, A. E. (1990). On the Use of Multiple Tokens on Ring Networks INFOCOM'90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration., IEEE**:** 15-22.

Kamrani, F. and R. Ayani (2007). Using On-Line Simulation for Adaptive Path Planning of UAVs. Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications.

Karamati, S., J. Young and R. Vuduc (2018). An energy-efficient single-source shortest path algorithm. International Parallel and Distributed Processing Symposium (IPDPS), IEEE**:** 1080--1089.

Keckler, S. W., W. J. Dally, B. Khailany, M. Garland and D. Glasco (2011). "GPUs and the future of parallel computing." IEEE Micro(5): 7--17.

Keville, K. L., R. Garg, D. J. Yates, K. Arya and G. Cooperman (2012). Towards fault-tolerant energy-efficient high performance computing in the cloud. International Conference on Cluster Computing (CLUSTER), IEEE.

Khanna, S., J. S. Naor and D. Raz (2002). Control message aggregation in group communication protocols. International Colloquium on Automata, Languages, and Programming, Springer**:** 135--146.

Kleinrock, L. (1975). Queueing systems. New York, Wiley.

Ladner, R. E. and M. J. Fischer (1980). "Parallel prefix computation." Journal of the ACM (JACM) **27**(4): 831--838.

Lau, W., R. Cherukuri, Y. Firstenberg, D. Louie, G. Anand and K. M. Woley (2013). Batching notifications to optimize for battery life. US, Microsoft Technology Licensing LLC

Lin, Y.-B. and E. D. Lazowska (1991). "A Time-Division algorithm for Parallel Simulation." ACM Transactions on Modeling and Computer Simulation **1**(1): 73-83.

Long, Y. and X. Hu (2017). "Spatial Partition-Based Particle Filtering for Data Assimilation in Wildfire Spread Simulation." ACM Transactions on Spatial Algorithms and Systems (TSAS) **3**(2): 5.

Lubachevsky, B. D. (1989). "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks." Communications of the ACM **32**(1): 111-123.

Luo, H. and S. Lu (2000). A topology-independent fair queueing model in ad hoc wireless networks. IEEE ICNP'00.

Madey, G. R., M. B. Blake, C. Poellabauer, H. Lu, R. R. McCune and Y. Wei (2012). Applying DDDAS Principles to Command, Control and Mission Planning for UAV Swarms. Proceedings of the International Conference on Compuational Science.

Maqbool, F., S. M. R. Naqvi and A. W. Malik (2017). Why to Redesign PDES Framework for Smart Devices: An Empirical Study. Summer Computer Simulation Multi-Conference. Seattle, Washington.

Merino, L., F. Caballero, J. R. Mart{\'\i}nez-De-Dios, I. a. n. Maza and A. i. b. Ollero (2012). "An unmanned aircraft system for automatic forest fire monitoring and measurement." Journal of Intelligent \& Robotic Systems **65**: 533 -- 548.

Microsoft-Edge. (2016, June 20). "Microsoft Edge Experiment: Battery Life." Retrieved 1 June, 2019, from https://www.youtube.com/watch?v=rjrxOOfi54k.

Mittal, S. (2014). "A Survey of Techniques For Improving Energy Efficiency in Embedded Computing Systems." International Journal of Computer Aided Engineering and Technology **6**(4): 440–459.

Möller, B. and C. Dahlin (2006). A first look at the HLA Evolved Web Service API. Euro Simulation Interoperability Workshop.

Muruganathan, S. D., D. C. F. Ma, R. I. Bhasin and A. O. Fapojuwo (2005). "A centralized energy-efficient routing protocol for wireless sensor networks." IEEE Communications Magazine **43**(3): 8-13.

Naldi, F. and A. Willig (2008). Batch delivery in wireless sensor networks. European Wireless Conference. Prague, Czech Republic IEEE.

Neal, S., R. Fujimoto and M. Hunter (2016). Energy consumption of data driven traffic simulations. Winter Simulation Conference (WSC), IEEE.

Neal, S., G. Kanitkar and R. Fujimoto (2014). Power Consumption of Data Distribution Management for On-Line Simulations. Principles of Advanced Discrete Simulation. Denver, Co., ACM**:** 197-204.

Ngu, A. H., Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z. Sheng. (2017). "IoT middleware: A survey on issues and enabling technologies." IEEE Internet of Things Journal 4 **4**(1): 1-20.

Nicol, D. M. (1988). "Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks." SIGPLAN Notices **23**(9): 124-137.

Nicol, D. M. (1993). "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations." Journal of the Association for Computing Machinery **40**(2): 304-333.

Nicol, D. M. and X. Liu (1997). The Dark Side of Risk. Proceedings of the 11th Workshop on Parallel and Distributed Simulation**:** 188-195.

Niu, L. and G. Quan (2004). Reducing both dynamic and leakage energy consumption for hard real- time systems. international conference on Compilers, architecture, and synthesis for embedded systems**:** 140–148.

NS-3_project. (2011). "NS3." Retrieved 1 June, 2019, from https://www.nsnam.org/.

Ou, Z., B. Pang, Y. Deng, J. K. Nurminen, A. Yla-Jaaski and P. Hui (2012). Energy-and cost-efficiency analysis of arm-based clusters. IEEE/ACM International Symposium on cluster, Cloud and Grid Computing (CCGrid), IEEE.

Park, H. and P. A. Fishwick (2010). "A GPU-based application framework supporting fast discrete-event simulation." Simulation **86**: 613--628.

Pathak, A., Y. C. Hu, M. Zhang, P. Bahl and Y.-M. Wang (2011). Fine-grained power modeling for smartphones using system call tracing. Sxth conference on Computer systems, ACM.

Peng, L., M. Silic and K. Mohseni (2015). "A DDDAS plume monitoring system with reduced Kalman Filter." Procedia Computer Science **51**: 2533--2542.

Pham, C. and C. Albrecht (1999). Optimizing message aggregation for parallel simulation on high performance clusters. International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). College Park, MD, USA, USA IEEE**:** 76--83.

Qualcomm_Technologies. (2015). "Trepn Profiler." Retrieved 1 June, 2019, from https://developer.qualcomm.com/software/trepn-power-profiler.

Quan, G. and X. Hu (2001). Energy efficient fixed- priority scheduling for real-time systems on variable voltage processors. Design Automation Conference**:** 828–833.

Rajovic, N., P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez and M. Valero (2013). Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC? Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. New York, NY, USA, ACM**:** 40:41--40:12.

Rajovic, N., A. Rico, J. Vipond, I. Gelado, N. Puzovic and A. Ramirez (2013). Experiences with mobile processors for energy efficient HPC. Design, Automation Test in Europe Conference Exhibition (DATE), 2013**:** 464-468.

Raya, M., A. Aziz and J.-P. Hubaux (2006). Efficient secure aggregation in VANETs. international workshop on Vehicular ad hoc networks, ACM**:** 67--75.

Rickert, M., K. Nagel, M. Schreckenberg and A. Latour (1996). "Two lane traffic simulations using cellular automata." Physica A: Statistical Mechanics and its Applications **231**(4): 534-550.

SAE. (2009). "SAE J2735 - Dedicated Short Range Communications (DSRC) Message Set Dictionary." Retrieved 1 June, 2019, from https://www.standards.its.dot.gov/Factsheets/Factsheet/71.

Saewong, S. and R. Rajkumar (2003). Practical voltage- scaling for fixed-priority rt-systems. IEEE Real- Time and Embedded Technology and Applications Symposium**:** 106–114.

Saleet, H. and O. Basir (2007). Location-Based Message Aggregation in Vehicular Ad Hoc Networks. Globecom Workshops. Washington, DC, USA IEEE.

Sanders P. and T. J.L. (2006). Parallel Prefix (Scan) Algorithms for MPI. Recent Advances in Parallel Virtual Machine and Message Passing Interface, EuroPVM/MPI 2006. M. B., T. J.L., W. J. and D. J., Springer, Berlin, Heidelberg**:** 49-57.

Schwartz, R. E. (2015). "How to Measure Power Consumption Using Free Software." Retrieved 1 June, 2019, from http://mostly-tech.com/2015/05/28/how-to-measure-the-power-consumption-of-your-mobile-app-using-free-software/.

Schwiebert, L., S. K. S. Gupta, P. S. G. Auner, G. Abrams, R. Iezzi and P. McAllister (2002). A biomedical smart sensor for the visually impaired. Sensors, IEEE.

Sengupta, S., M. Harris, M. Garland and J. D. Owens (2011). Efficient Parallel Scan Algorithms for many-core GPUs. Scientific Computing with Multicore and Accelerators. J. Kurzak, D. A. Bader and J. Dongarra, Chapman & Hall/CRC Computational Science**:** 413–442.

Sengupta, S., M. Harris, Y. Zhang and J. D. Owens (2007). Scan primitives for GPU computing. Graphics Hard-ware 2007**:** 97--106.

Shehabi, A., S. J. Smith, D. A. Sartor, R. E. Brown, M. Herrlin, J. G. Koomey, E. R. Masanet, N. Horner, I. L. Azevedo and W. Lintner (2016). United States Data Center Energy Usage Report, Lawrence Berkeley National Laboratory.

Shi, W., K. Perumalla and R. Fujimoto (2003). Power-aware State Dissemination in Mobile Distributed Virtual Environments. Workshop on Parallel and Distributed Simulation, San Diego.

Shueh, J. (2017). "Atlanta Smart Corridor Evolves into a Springboard for Autonomous Vehicles, IoT and More." Retrieved 1 June, 2019, from

http://smartatl.atlantaga.gov/index.php/blog-post/atlanta-smart-corridor-evolves-into-a-springboard-for-autonomous-vehicles-iot-and-more/.

Silberschatz, A., P. B. Galvin and G. Gagne (2006). Deadlocks. OPERATING SYSTEM PRINCIPLES, Wiley India Pvt. Limited**:** 245--271.

Sill, S. (2016). "DSRC: The Future of Safer Driving." Retrieved 1 June, 2019, from https://www.tampacvpilot.com/wp-content/uploads/2016/12/dedicated-short-range-communications-factsheet.pdf, https://www.its.dot.gov/factsheets/dsrc_factsheet.htm.

Soini, M., J. Van Greunen, J. Rabaey and L. Sydanheimo (2007). Beyond sensor networks: Zuma middleware. Wireless Communications and Networking Conference (WCNC), IEEE.

Sokolowski, J. A. (2008). Simulation: Models That Vary over Time. The Practice of Modeling and Simulation: Tools of the Trade, Modeling and Simulation in the Medical and Health Sciences. J. A. Sokolowski and C. M. Banks, John Wiley & Sons, Inc.**:** 23-33.

Souto, E., G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa and C. Ferraz (2004). A message-oriented middleware for sensor networks. workshop on Middleware for pervasive and ad-hoc computing.

Stanisic, L., B. Videau, J. Cronsioe, A. Degomme, V. Marangozova-Martin, A. Legrand and J.-F. Mehaut (2013). Performance analysis of HPC applications on low-power embedded platforms. Design, Automation Test in Europe Conference Exhibition (DATE), 2013**:** 475-480.

SUMO. (2005). "SUMO." Retrieved 1 June, 2019, from http://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/.

SUMO_application_network. (2005). "SUMO application network." Retrieved 1 June, 2019, from http://sumo.dlr.de/wiki/Tutorials/quick_start.

Sung, S., D. Noh and J. Park (2013). Development of Reliable V2V System Based on WAVE. 23rd International Technical Conference on the Enhanced Safety of Vehicles (ESV).

Tiwari, V., S. Malik, A. Wolfe and M. T.-C. Lee (1996). Instruction level power analysis and optimization of software. Technologies for wireless computing, Springer**:** 139--154.

Tyer_labs and Rappsilber_labs. (2014). "BoxPlotR." Retrieved 1 June, 2019, from http://boxplot.tyerslab.com/, http://shiny.chemgrid.org/boxplotr/.

Unsal, O. S. (2008). System-Level Power-Aware Computing In Complex Real-Time and Multimedia Systems. Doctor of Philosophy Doctoral Dissertation, University of Massachusetts.

Wang, J. J. and M. Abrams (1992). Approximate Time-Parallel Simulation of Queueing Systems with Losses. Winter Simulation Conference.

Want, R. (2014). "The power of smartphones." IEEE Pervasive Computing **13**(3): 4.

Williams, S., A. Waterman and D. Patterson (2009). "Roofline: an insightful visual performance model for multicore architectures." Communications of the ACM **52**(4): 65--76.

Yan, S., G. Long and Y. Zhang (2013). StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchro-nization. Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Pro-gramming**:** 229--238.

Yu, Y., B. Krishnamachari and V. K. Prasanna (2004). "Issues in designing middleware for wireless sensor networks." IEEE network **18**(1): 15-21.

Zhang, L., B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao and L. Yang (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM.